Software and Systems Modeling (2024) 23:163–193
https://doi.org/10.1007/s10270-023-01112-6

REGULAR PAPER

Scientific workflow execution in the cloud using a dynamic runtime model

Johannes Erbel[1] · **Jens Grabowski**[1]

Received: 17 December 2021 / Revised: 5 May 2023 / Accepted: 8 May 2023 / Published online: 23 June 2023

Abstract
To explain specific phenomena, scientists perform a sequence of tasks, e.g., to gather, analyze and interpret data, forming a scientific workflow. Depending on the complexity of the workflow, scientists require access to various kinds of tools, applications and infrastructures for individual tasks. Current approaches are often limited to managing these resources at design time, requiring the scientist to preemptively set up applications essential for their workflow. Therefore, a dynamic provisioning and configuration of computing resources are required that fulfills these needs at runtime. In this paper, we present a dynamic runtime model that couples workflow tasks with their individual applications and infrastructure requirements. This runtime model is used as a knowledge base by a model-driven workflow execution engine orchestrating the sequence of tasks and their infrastructure. We exhibit that the simplicity of the runtime model supports the creation of highly tailored infrastructures, the integration of self-developed applications, as well as a human-in-the-loop allowing scientists to monitor and interact with the workflow at runtime. To tackle the heterogeneity of cloud provider interfaces, we implement the workflow runtime model by extending the Open Cloud Computing Interface cloud standard, which provides an extensible data model as well as a uniform interface to manage cloud resources. We demonstrate the applicability of our approach using three case studies and discuss the benefits of the runtime model from a user and system perspective.

Keywords Runtime model · Workflow · Cloud · OCCI

1 Introduction

The benefits of performing scientific calculations on heterogeneous and distributed computing resources allow scientists to efficiently gather and analyze data at scale from various sources for various domains [1]. Often these calculations are put into a sequence, forming a *Scientific Workflow* (SWF) [2], which can be processed by a corresponding *Scientific Workflow Management System* (SWFMS) [3]. Depending on the complexity of the workflow and the complexity of the computations to be performed, the individual workflow tasks require access to different kinds of tools, applications and infrastructures. Therefore, the deployment and especially the combination of different applications running in parallel to the workflow are cumbersome and often lead to a permanent deployment of rarely used resources. To tackle this issue, we are proposing a concept and a prototypical implementation that combines workflow, runtime model and cloud orchestration techniques. Using the abstract nature of *Model-Driven Engineering* (MDE), we strive to reduce the complexity of using and combining distributed tools and frameworks, while providing scientists with the means to integrate self-developed or emerging computation frameworks into their workflow. This enables the automated creation of individual infrastructure for specific tasks required by the scientist, with the runtime model providing the transparency to inspect and react to intermediate results at runtime.

Meanwhile, many SWFMSs have been adapted to run in the cloud to utilize its scalable and on demand resources. However, many of them do not grant direct access to the infrastructure layer. Therefore, scientists are often restricted to the applications and infrastructures available to them at design time, requiring them to preemptively set up all the computation clusters and applications essential for their

Communicated by Jeff Gray.

✉ Johannes Erbel
johannes.erbel@cs.uni-goettingen.de

Jens Grabowski
grabowski@cs.uni-goettingen.de

[1] University of Goettingen, Institute of Computer Science, Goettingen, Germany

workflow. This design time setup may drastically increase the resource consumption and thus cost for executing a workflow, as, e.g., specific resources are only required by a specific experiment or a single task within the workflow [4].

Similar to Qasha et al. [5], we address these issues by combining cloud orchestration techniques with workflow management. Additionally, we make use of model-driven techniques to reflect the scientific workflow, as well as its deployed infrastructure, within a causally connected *runtime model*. This connection ensures that changes to the model are directly propagated to the system and vice versa [6, 7]. In our approach, we mainly connect the workflow runtime model to a cloud infrastructure. Furthermore, we connect the model to a simulation environment that supports the development of the workflow without the need to provision actual cloud resources. Overall, according to a recent literature study by Bencomo et al. [8], the utilization of runtime models for workflows is not well investigated in the current state of the art.

Due to the simplicity of our approach and the abstract representation of the workflow and infrastructure, scientists can easily access distributed resources and focus on how they should dynamically change throughout the workflow execution. Furthermore, using our approach scientists can directly monitor the state of the workflow and the infrastructure while the causal connection allows to directly interact with the workflow model by design. Combined, this supports human-in-the loop allowing, e.g., to respond to intermediate results at runtime addressing one of the remaining challenges for large-scale scientific workflows according to Deelman et al. [9]. A further challenge that we tackle is the *dynamic* provisioning of distributed resources by granting our model direct access to infrastructure configurations. This allows to manage arbitrary compute, storage and network resources at runtime.

Due to the success of cloud computing, multiple providers emerged granting access to on demand computing resources. This led to a multitude of different interfaces, making it difficult to switch from one provider to another [10]. To foster the reusability and portability of our approach, we extend the *Open Cloud Computing Interface* (OCCI) [11], a cloud standard by the *Open Grid Forum* (OGF). This standard specifies an extensible data model for cloud resources as well as an uniform interface to manage them.

In a position paper [12], we initially described the concept of tailoring workflow models to individual infrastructural and application needs. We further discussed the reflective capabilities of this runtime model in [13] where we demonstrate a light-weight and model-driven workflow management system built around the runtime model.

In this paper, we investigate the flexibility of a workflow runtime model by extending the runtime metamodel to incorporate loops and parallelization. We discuss how the reflective capabilities of the causally connected runtime model can be used to dynamically adjust and provision cloud resources at runtime. We evaluate our approach using three case studies in which we modeled and executed scientific workflows from different domains, namely big data analysis, dynamic simulations and software repository mining. Based on these studies, we discuss the applicability of runtime models to dynamically manage and deploy workflows from different perspectives. In order to perform these studies, we developed a SWFMS prototype that integrates into an existing OCCI ecosystem, which are both publicly available.[1] With this work, we provide the following new contributions:

- The reflection and management of loops in the workflow runtime model,
- The automatic replication of resources for parallel workflow tasks and
- An extended evaluation of the runtime model for workflow execution.

The remainder of this paper is structured as follows: Section 2 covers foundations of scientific workflows, runtime models and cloud computing. Section 3 introduces the related work. Section 4 describes our workflow runtime model concept, as well as the SWFMS built around it. Section 5 introduces the implementation of the concept which is used to perform the case studies described in Sect. 6. The result of designing and executing the case studies is discussed in Sect. 7 including the identified threats to validity. In Sect. 8, we provide an overall conclusion and an outlook into future work.

2 Foundations

In the following, the individual foundations of the approach are introduced. This includes scientific workflows, models at runtime, as well as basics of cloud computing and respective cloud models.

2.1 Scientific workflows

In general, a workflow is a sequence of computational tasks to be executed. This sequence is typically expressed as a *Directed Acyclic Graph* (DAG) or a *Directed Cyclic Graph* (DCG), depending on whether the workflow contains loops [2]. In literature and industry, a multitude of languages exist to describe workflows, e.g., activity diagrams of the *Unified Modeling Language* (UML) [14] which describe the behavior of a software system over a sequence

[1] All URLs have been last retrieved on 05/05/2023. https://gitlab.gwdg.de/rwm/smartwyrm

of actions, or the *Business Process Model and Notation* (BPMN) [15] used to describe business processes. Compared to these workflows, scientific workflows are used to model and run scientific experiments [3]. The distinction between both workflow scenarios is blurry and differs mainly in the goals and priorities of the workflow [16]. According to Ludäscher et al. [17], control flow-oriented workflows are more oriented towards business workflows, while data flow-oriented are more often used for scientific reasons. We stick to the notion of a scientific workflow, as our approach focuses on tailoring workflows towards specific infrastructural needs supported by a runtime model which we evaluate with case studies from the scientific domain. Still, due to blurry distinction the presented approach can be applied to data or computational driven workflows.

To interpret modeled workflows and automate their execution, a SWFMS is required [9]. Depending on the computation infrastructure utilized, workflows can be further divided into *in situ workflows* and *distributed workflows* [9]. While in situ workflows exchange information over internal storage or internal networks of *High Performance Computing* (HPC) systems, our approach represents a distributed workflow concept in which tasks are loosely coupled and executed on distributed resources such as grid and cloud systems.

2.2 Models at runtime

A *model* is an abstract representation of a system that is either prescriptive or descriptive, which allows predictions to be made [18]. Within the model, the amount of information about the real-world system is reduced to a minimum to be easy to understand while maintaining its descriptiveness [19]. Each model is an instance of a *metamodel* defining a set of elements available to build a model and therefore serves as "a language of models" [20].

A *runtime model* is also based on a metamodel and is used to abstract a system. Furthermore, runtime models possess a causal connection to the abstracted system [6, 7]. Therefore, changes in the runtime model are directly propagated to the system and vice versa allowing it to depict a system's runtime state. Using this strong connection to the system, a runtime model does not only support the system's monitoring, but also allows to control and reason about it [21].

As runtime models are based on a metamodel, MDE techniques, such as *model transformations* [22, 23], can be applied, which allow to adjust the system on a high level of abstraction. Moreover, the descriptive nature of a runtime model allows it to serve as a knowledge base for self-adaptive systems [7] which, e.g., utilize *Monitor–Analyze–Plan–Execute–Knowledge* (MAPE-K) control loops [24]. In our approach, we present a runtime model concept allowing scientists and management systems to observe and interfere with the workflow execution. We establish a causal connection of the model and the distributed computing resources in order to orchestrate the required architecture of each individual task within the workflow.

2.3 Cloud computing

At its core cloud computing is a paradigm in which virtualized chunks of a large pool of computing resources can be rented from a provider [25]. The sheer amount of resources that can be dynamically rented on demand removes the need to plan ahead of time. This in turn allows to quickly react to changing computation needs and requirements, which results in a multitude of approaches to schedule these resources. While multiple types of service models can be offered by a cloud provider, e.g., *Software as a Service* (SaaS), *Platform as a Service* (PaaS) and *Infrastructure as a Service* (IaaS), our approach focuses on the IaaS granting full access to infrastructural resources.

Over the years, multiple providers emerged resulting in different *Application Programming Interfaces* (APIs) to access the cloud services and rent resources, making it difficult to switch from one provider to another [10]. To tackle this issue, standards got created, like the *Topology and Orchestration Specification for Cloud Applications* (TOSCA) [26], developed by the *Organization for the Advancement of Structured Information Standards* (OASIS) and the *Open Cloud Computing Interface* (OCCI) [11], developed by the *Open Grid Forum* (OGF). Within our approach, we focus on the utilization of cloud resources due its elastic, flexible and on-demand capabilities, while creating an implementation that is built around the OCCI standard.

3 Related work

Besides the presented approach, many efforts have been made to model and process workflows. In the following, we provide an overview of existing scientific workflows and management systems, as well as approaches combining cloud and workflow orchestration.

According to Deelman et al. [9], most SWFMS rely on scripting languages or even provide a graphical user interface to define and manage tasks as well as their dependencies. These, e.g., include Swift [27], Tigres [28], Kepler [29], Trident [30], Weaver [31], Triana [32], Pegasus [33, 34], Galaxy [35], Taverna [36, 37], VisTrails [38], SCIRun [39], Wings [40], and AVS [41]. Even though many languages exist, most focus on the workflow description at design time and are only used as input for a SWFMS without considering infrastructure descriptions or runtime representations.

The concept of SWFMS exists for a long time with a lot of implementations using the benefits of distributed computing platforms such as grid computing. Among oth-

ers these comprise, but are not limited to DAGMan [42], ICENI [43], GrADS [44], Grid-Flow [45], UNICORE [46], Gridbus workflow [47], Askalon [48] and Chiron [49]. Also, concepts exist in the literature that address the need for spatiotemporal models that combine software and workflow components [50, 51]. Many of the named SWFMS are meanwhile adapted to use cloud resources, e.g., Pegasus [34] or Taverna [37]. Often the cloud resources are utilized to scale a preconfigured workflow management cluster on demand. Furthermore, approaches exist that utilize the cloud environment to spawn workflow middleware on demand [52] or deploy scaling architectures for the execution of workflows [53]. While each management system has their own benefits and drawbacks, many of them exist before the rise of infrastructure as code. Therefore, often no direct access to infrastructural resources and application deployments is provided which allows the user to create tailored workflows with runtime management capabilities.

In [54], the authors describe the deployment of HyperFlow [55], a SWFMS for distributed systems, on an automatically provisioned Kubernetes cluster to manage and scale a container infrastructure for the workflow. In [56] HyperFlow is coupled with PaaSage to foster the utilization of cloud and HPC resources within a single workflow. PaaSage itself is a cloud resource manager based on the *Cloud Application Modelling and Execution Language* (CAMEL) [57], a runtime metamodel to manage and scale cloud applications. Even though this approach builds upon a runtime model for cloud applications, the workflow layer is not reflected in the model.

In addition to the formerly mentioned systems, approaches exist that are focused on coupling specific infrastructure to workflow tasks. Qasha et al. [5] extend the TOSCA standard to ensure the reproducibility of workflows in the cloud utilizing the benefits of containerization. Also, Weder et al. [58] utilize TOSCA to describe workflows and their required cloud compositions in a self-contained manner which gets deployed via OpenTOSCA [59]. Even though the TOSCA standard is extended, no model representation is considered at runtime that allows the scientist to observe and manipulate the workflow. While most of the related work do not consider the utilization of runtime reflections, some approaches exist heading into this direction. In the approach described in [60], a reflective middleware is presented that gathers information from a workflow model at runtime to optimize infrastructure deployments. Here, annotations are used to describe the workflow's resource and deployment requirements. The annotations, however, do not couple workflow tasks with modeled platform elements directly and therefore limit runtime model capabilities for deployed applications.

In [61], Colonnelli et al. present StreamFlow, a workflow approach that combines capabilities provided by cloud and HPC using container orchestrators to deploy dedicated infrastructures. Hereby, they couple deployment descriptions with individual workflow tasks within YAML file that conforms with the *Common Workflow Language* (CWL) interface. While StreamFlow presents a matured approach to couple heterogeneous execution environments and the accompanied complex dataflows, it does not highlight runtime reflections. Due to the utilization of YAML, the running workflow heavily relies on a design time workflow configuration. In turn, reflective capabilities of the workflow are limited, making runtime changes by the user complicated, like increasing the amount of desired parallelization based on intermediate results. Finally, approaches like Mashup [62] couple emerging trends in cloud computing, like serverless-computing, with scientific workflows to further mitigate under-utilization and over-provisioning of computing resources. While Mashup optimizes resource management in scientific workflows, our approach focuses on the abstraction and reflection of scientific workflows and infrastructure in model-driven manner with the possibility to use an underlying model-driven toolchain that is based on the OCCI standard to integrate upcoming trends.

As reflected by the sheer amount of workflow approaches, there is a huge interest in utilizing the elastic capabilities of cloud computing. However, currently there is no approach that allows scientists to model, execute and reflect workflows with arbitrary and shifting infrastructure requirements for individual tasks in a dedicated runtime model. In our approach, we combine the benefits of MDE and cloud orchestration techniques allowing scientists to attach and model their own individual requirements to tasks within their workflow. Furthermore, we implement our approach by extending OCCI to use a standardized language for cloud deployments. While other SWFMS concepts focus on the execution of workflows described via design time artifacts, our approach focuses on the causal representation of the workflow and its infrastructure within a runtime model. Herewith, we allow scientists to directly monitor and interact with the workflow while giving a SWFMS, as well as other systems, a single point of access to manage the workflow and its infrastructure removing the need for preconfigured computation cluster.

4 Approach

Similar to other SWFMS concepts, our approach can be separated into the language with which the workflow can be modeled and the engine interpreting it. In Sect. 4.1, we provide an overview of our approach, followed by a description of the used workflow runtime metamodel in Sect. 4.2 and its corresponding engine in Sect. 4.3. For the remainder of this article, we introduce a legend for the following figures, shown in Table 1. The table maps icons to utilized Workflow, Platform and Infrastructure elements which are explained

Table 1 Legend of resources and their corresponding icons

Workflow	Platform	Infrastructure
Task	Application	Compute
Decision	Component	Network
Loop	Sensor	Storage

together with the metamodel in Sect. 4.2. These icons are used within a graph-based structure serving as a visual notation for our approach and implementation.

4.1 Overview

To allow for an automated workflow execution, a design time model describes the Workflow ①, as shown in Fig. 1, which serves as input for the Workflow Engine ②. This engine then dynamically provisions the modeled Infrastructure ③ resources and executes the tasks.[2]

Each task within the Workflow ① is coupled to its individual requirements which, compared to related approaches, allows for dynamic and arbitrary infrastructures to be used throughout the workflow. When creating the workflow model, the Scientist models the tasks and their control and dataflow, while a Cloud Architect models the underlying infrastructure including deployed applications. Hereby, applications can be designed in a reusable manner to be incorporated within multiple workflows easing the utilization of the approach for the scientist. For example, the Workflow in Fig. 1 contains a task A which requires a single Storage and one Compute node, e.g., a *Virtual Machine* (VM). This task only requires few computing resources and may be used to gather data, while task B requires a computation cluster to be deployed, e.g., to analyze the gathered data. It should be noted that applications can also be modeled hosted on the depicted infrastructure, which we omitted from this figure for clarity.

The Workflow Engine ② takes the designed Workflow as input. The engine and the approach itself are build around a Runtime Model serving as knowledge base and interface to deploy required resources. Compared to the design time Workflow, the Runtime Model is directly synchronized with the Infrastructure causally connecting the state of each task, as well as currently deployed resources. Due to the abstract representation of the Runtime Model, the Scientist can observe and manipulate the workflow at runtime, e.g., by adjusting decision-making points based on intermediate

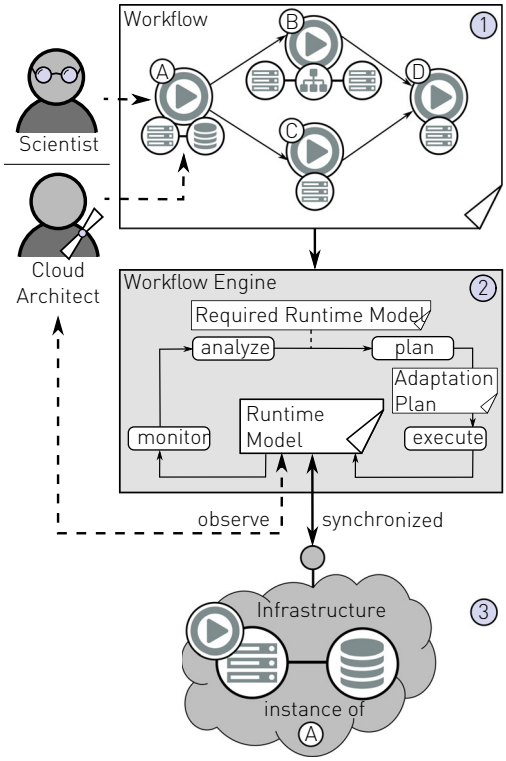


Fig. 1 Model-driven scientific workflow orchestration using a runtime model as knowledge base

results. To perform the workflow, the engine uses a MAPE-K loop in which we monitor the current state of the system through the runtime model. Next, we analyze how the information of the design time and runtime information of the workflow can be merged to into a Required Runtime Model describing the infrastructure state currently required by the workflow. Based on the resulting model, we plan how to reach the desired state by generating an Adaptation Plan. Finally, this plan is executed adapting the Runtime Model.

The underlying Infrastructure ③ is adapted, as soon as the Runtime Model is changed due to their causal connection. In case of the example shown in Fig. 1, task A is currently running with its corresponding infrastructure and applications being deployed. The completion of task A is reflected in the Runtime Model allowing the Workflow Engine to identify that in the next workflow iteration tasks B and C can be executed. This results in their required infrastructure resources to be provisioned next. It should be noted that the causal connection of the Runtime Model to the Infrastructure is not limited to cloud environments. For testing purposes, e.g., the causal connection may be completely disabled allowing scientists to simulate the execution of workflows on a model-based level supporting the development process.

[2] Legend shown in Table 1, page 5.

4.2 Runtime workflow metamodel

To reflect the state of the workflow and state of the infrastructure within a runtime model, a metamodel is required providing access to infrastructure and application resources (e.g., compute, storage, networks, components and applications), as well as workflow elements (e.g., tasks, decisions, loops). To conform to already existing approaches, we stick to the typical notion of a workflow with each element being either a node or an edge maintaining the form a DAG/DCG. Figure 2 depicts a high-level concept of the runtime workflow metamodel using the icons of Table 1. Furthermore, we omit attributes and actions that can be triggered on the individual entities for clarity. We separate the metamodel elements into five layers: Graph ①, Workflow ②, Platform Dependency ③, Platform ④ and Infrastructure ⑤. A Graph ① describes an abstract concept formed by abstract *Node* and *Edge* elements. Each *Edge* connects two nodes by referencing its source and target. At its core, each element within the metamodel inherits either from *Node* or *Edge*. We omitted the visualization of inheritance relationships for clarity and replaced them with white rectangles being *Node* specializations and grey rectangles being *Edge* specializations.

In case of the Workflow ② layer, a Task represents a node type and is used, e.g., to describe a program or computation to be performed. A TaskDependency, on the other hand, represents an edge between two tasks and is used to describe a control or dataflow between them. In addition to the base task type, we introduce Decision and Loop nodes representing specialized versions of a Task used for decision-making processes.

Also, the Platform ④ layer can be adapted to the concept of a graph, as shown in the OCCI platform extension [63]. For example, Application and Component are resources that can be modeled as nodes and connected via specialized edges. We refer to an Application as a larger software system that consists of multiple smaller units represented by Component instances. To describe this relationship, a ComponentLink can be used. Furthermore, this edge can used as a connection between two Component nodes to describe dependencies between them. One example for an application is a larger Sensor application that consists of several monitoring devices deployed on different machines [64]. Finally, this layer consists of a PlacementLink which connects resources from the Platform and Infrastructure layer [65]. Each PlacementLink is part of a Component with its target attribute describing the resource it gets deployed on.

The Infrastructure ⑤ layer describes the computing resources used to host applications and perform tasks on. A small subset of infrastructure resources is represented by Compute, Storage and Network nodes as described in the OCCI infrastructure extension [66]. A Compute node represents an abstract concept that can be used to describe, e.g., a VM or a deployed container. Also for this layer, specialized edges can be used to describe the interconnections of infrastructural resources. For example, a Compute node can be attached to a Storage using a StorageLink or to a Network using a NetworkInterface.

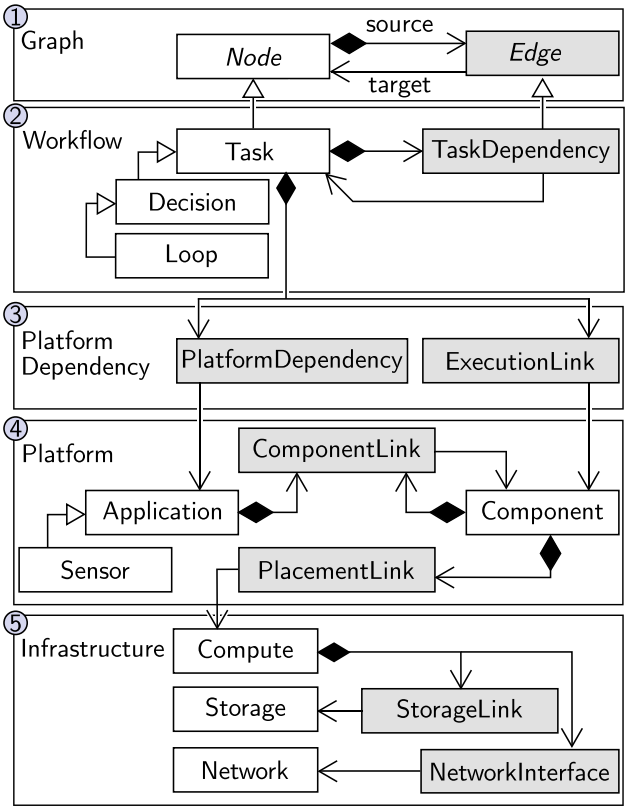


Fig. 2 Runtime workflow metamodel with node types in white and edge types in grey

Multiple approaches and metamodels already exist that describe graph, workflow, platform and infrastructure elements. In our approach, we introduce a Platform Dependency ③ layer interconnecting workflow task sequences with infrastructure resources to create an overall workflow runtime model. As this layer is solely used for interconnection purposes it consists of two edge types. A PlatformDependency connects a Task to a platform resource, e.g., an Application, and describes the platform requirements that need to be fulfilled in order to execute the task. Hereby, the infrastructure resources to be provisioned can be traced via connections to Component instances and the Compute nodes they are placed on. Similar to application deployments, a task requires an executable component that needs to be deployed on one of the distributed resources. To model this deployment, an ExecutionLink is used connecting a Task to a Component instance representing the computation or program to be executed.

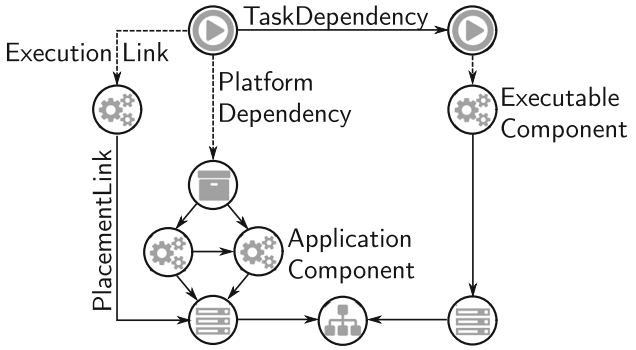


Fig. 3 Example instance of the workflow runtime model

By combining the layers, the workflow runtime metamodel (Fig. 2) allows creating models following the schema shown in Fig. 3.[3] Consecutive tasks are sequenced by a TaskDependency. Each task has an ExecutionLink to its Executable Component and a PlatformDependency to the application required to run the task, visualized as dashed lines. As both task and application nodes can be connected to components, we differentiate between an Executable Component and an Application Component. An Executable Component describes the logic, program or computation of the task to be triggered when started. An Application Component implement parts of the application that can be managed over the application node, e.g., to manage the deployment of an application which consists of a master and several worker nodes. In this case, the application consists of two Application Component instances interconnected with component links. Independent of the type, each component is connected via a PlacementLink to a compute node it shall be hosted or deployed on which are attached to a network node using a network interface link.

4.2.1 Runtime model capabilities

Each node within the runtime model possesses a state, as well as attributes over which it can be described. Depending on the layer, the attributes, for example, may describe the amount of cores of a VM possesses or a specific version of a component to use. In addition to attributes, actions exist that allow to access these resources and progress through their states. For example, infrastructure nodes may be described via an active and inactive state that can be started and stopped [66], while platform components may be deployed, configured and started [63, 65]. In case of tasks, the states are depicted in Fig. 4 in form of a *Finite State Machine* (FSM), visualized using UML. The FSM for task states comprises a scheduled, active, inactive, skipped and finished state, as well as actions that allow to transition between those. It should

[3] Legend shown in Table 1, page 5.

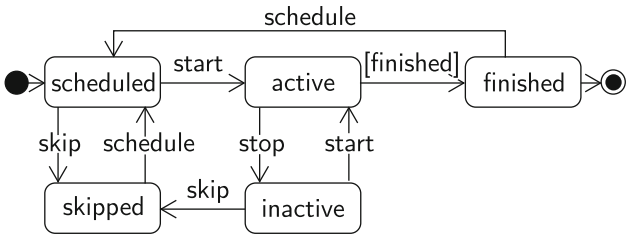


Fig. 4 Finite state machine representing the states of a task

be noted that an additional error state exists that is reached on failed state transitions which is not displayed for clarity. Each task starts within a scheduled state indicating that it needs to be processed. The execution of the tasks itself can then be triggered via a start action and stopped via a stop action, as long as the task is not finished. Triggering the start action of a task results in the deployment and execution of its executable component performing the computation task on the compute node it is hosted on. Once a task is finished with its execution, it may be rescheduled, which is especially important for the execution of loops. Finally, a scheduled or inactive task can reach the skipped state which is utilized for decision-making processes indicating which control flow to follow. This process and its required elements are described in the following.

4.2.2 Decision-making

One major requirement of a workflow system is to dynamically decide on a specific control flow, or task sequence, to follow. Hereby, the decision is based on intermediate results or information gathered at runtime. To allow for such decision-making processes, we introduce a Decision element as well as a ControlflowGuard, as shown in Fig. 5. Even though the concept of both elements is well known, the utilization within an infrastructure aware runtime model is to the best of our knowledge not discussed in the literature. The Decision element is responsible to gather, store and process runtime information to decide which control flow to follow within the workflow. As the gathering and processing of a Decision represents an individual step within the workflow that needs to be executed it inherits from Task. Due to this relation, a Decision has access to the same characteristics as a Task, including its connection to the platform and infrastructure layer. In case of a decision node the executable typically represents a Sensor which is used to reflect monitoring data in the runtime model [64]. A Sensor is a specialized version of an application, as it contains components used to gather, process and publish runtime information to the model, e.g., to read and reflect log information required for decision-making. Compared to a typical sensor that con-

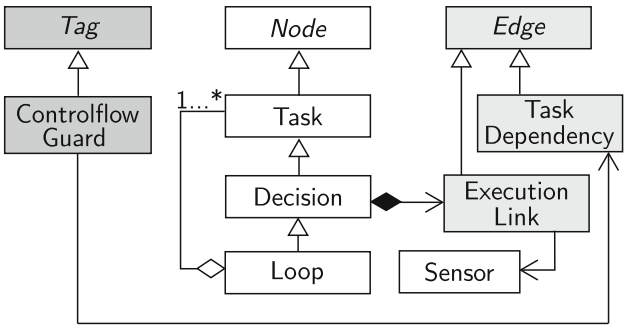


Fig. 5 Runtime workflow decision and loop concept

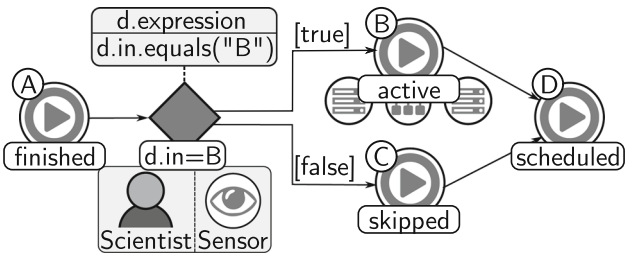


Fig. 6 Workflow decision example

tinuously produces monitoring results, the Decision stops the Sensor once it has received its decision input.

The gathered information is then checked against a ControlflowGuard, which is a specialization of a *Tag*. In general, a *Tag* instance extends the notion and attributes of an entity, which in case of the ControlflowGuard can be dynamically attached to a TaskDependency. Hereby, if the gathered runtime information does not fulfill the condition of the guard, the task connected by the TaskDependency edge is propagated to the skipped state, as well as each subsequent task that cannot be reached otherwise. Therefore, only fulfilled guards remain in the scheduled state. This allows the SWFMS to pursue the goal of bringing each task into either a finished or skipped state.

Figure 6 exemplifies a workflow decision-making model.[4] It should be noted that this example can be referred to the general schema of attaching workflow tasks to infrastructure resources as shown in Fig. 3. Apart from the schema, one major difference between a Task and a Decision is the ability to store runtime information within an attribute (d.in) to be processed. This attribute is either filled automatically by an executable attached to the decision node or manually over a human input. While an automatic decision-making process requires a specialized task, a manual adjustment of the control flow requires the Scientist to only affect the decision input attribute. Furthermore, a decision expression attribute (d.expression) allows the scientist to describe how the runtime information stored within the decision input is evaluated, e.g., to check whether the gathered value is above a certain threshold.

In the visualized example, the decision input d.in is set to B either manually by the Scientist or directly by the Sensor. When the decision node is triggered, the decision expression (d.in.equals("B")) is evaluated which in this case sets the decision result to true. The result is then checked against the control flow guards ([true] and [false]) of each connected task dependency. Here the evaluation leads to task C being skipped, while task B remains scheduled. Also task D remains scheduled as it can still be reached over the control flow of task B. At this point, the decision node has completed its processing bringing it into the finished state. Therefore, all tasks prior to task B are now finished allowing its infrastructure to be provisioned and the task itself to be started resulting in an active state of the task as shown in the figure.

4.2.3 Loop reflection

While a task may internally loop over a set of items, it does not allow to reflect the current iteration it is processing within the runtime model. Moreover, in order to loop over a set of modeled tasks with arbitrary infrastructure requirements, we introduce a loop element which reflects all the information required to manage a DCG. As shown in Fig. 5, we define a Loop as a Decision specialization that contains a sequence of Task nodes to be executed while a certain condition holds.

Throughout each iteration, a Loop node needs to decide whether to perform a loop iteration or not using the same mechanics described in the decision-making process, i.e., using a decision input and expression. If a loop iteration is required, the Loop is set to an active state describing that it is currently processing its contained tasks. Typically, each task requires each previous task to be finished, however, the first task within a loop only requires the loop to be in an active state. This allows to clearly reflect that a Loop is running in the runtime model. As soon as the final task during a loop iteration has reached the finished state, the loop performing the decision-making process for the next iteration is notified. If another iteration is required each looped task is rescheduled, allowing them to be processed again. Otherwise, the loop as well as its containing tasks remain in the finished state. If no iteration is required at all, each looped task is put into a skipped state, similar to the logic of a Decision node. Even though no iteration is performed, the loop is executed successfully reaching the finished state allowing to perform follow up tasks.

During each iteration, a loop infuses its contained tasks with iteration-specific information allowing the tasks, e.g., to iterate over a set of items. Our approach allows to model spe-

[4] Legend shown in Table 1, page 5.

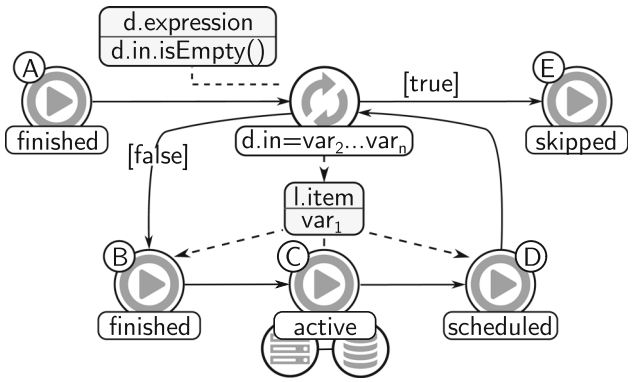


Fig. 7 Workflow example with an active loop

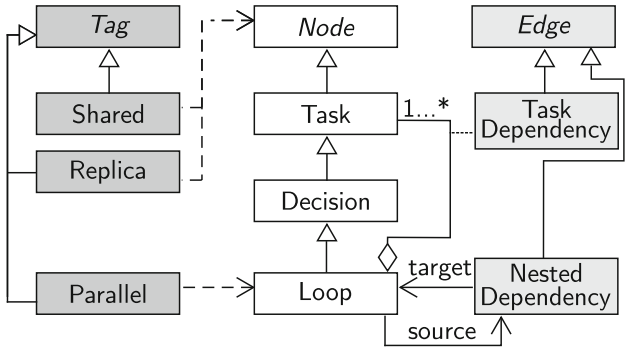


Fig. 8 Runtime workflow loop and parallelization concept

cialized loops, such as for each and while loops. Depending on the kind of loop, different information need to be reflected. For example, in a while loop the current iteration performed can be injected into each looped task, while in a for-each loop the individual tasks get infused with information about the current item that gets processed.

Figure 7 depicts an example runtime workflow containing a loop that iterates over a set of items ($var_2...var_n$) with var_1 being currently processed.[5] To iterate over the items, the runtime information is gathered utilizing the same procedure provided by decision nodes. As long as the decision input (d.in) is not empty, the d.expression evaluates to false resulting in the task E to be skipped for this iteration and B to be scheduled which allows the looped tasks to be executed. Task B represents the first task of the loop which, in case of Fig. 7, is already finished resulting in task C being active while task D is scheduled waiting for C to be finished. During each iteration, the loop evaluates whether the decision expression still holds for the current iteration. If an additional iteration is performed, the next item, in this case var_2, is removed from the decision input and provided to each looped task. This way the decision expression evaluates to false when each item has been processed.

4.2.4 Parallelization of loops

In general, we couple the logic of parallelizing loops into the runtime model itself, which allows to create a certain amount of replicas modeled or changed by the user at runtime. When a parallel loop is triggered, the runtime model spawns nested loops with replicated tasks to be processed by the system. This way, adjustments of the parallelization values directly affect the runtime model leaving the infrastructure provisioning and scheduling process to the SWFMS. For example, the modeled infrastructure for a task could simply be duplicated for each newly spawned task.

To parallelize the workload of a loop, we adopt the concept of OpenMP,[6] an API for parallelization in C, C++ and Fortran, and simply annotate a loop with parallelization information. As shown in Fig. 8, a Loop can be annotated with a Parallel tag which allows to define the amount of parallel executions to be created at runtime. When the start action of a loop is triggered, this value is used to determine how many times the modeled loop has to be duplicated. According to the defined amount, a set of nested loops are created within the runtime model, which get connected to the originally modeled loop via a NestedDependency link. This loop is now responsible to reflect the overall state of the originally planned loop. While one of the created nested loops contains the original set of modeled tasks, the other nested loops contain duplicated versions of the tasks tagged as Replica. Within this tag, a reference to the original is stored. Furthermore, each replica tag stores a reference to the specific replica group it belongs to which is led by a nested loop. These references are especially used to derive the required architecture for the replicated tasks at runtime. Finally, the items to be processed are split up among the nested loops, which then distribute the workload to their looped tasks. By default, the items are split in half with the possibilities to extend the separation of workload using different strategies. To separate elements to be replicated within parallelized loops from elements not to be replicated, we introduce a Shared tag. This tag allows scientists to define which resources should be shared during parallel sections of their workflow. Moreover, this tag helps the scheduling mechanisms to understand that this element is shared among tasks within the loop. One example use case for the tag is, e.g., the utilization of a database application storing the information of all tasks within the workflow. In this case the application and its underlying infrastructure should not be replicated even though it may be part of a parallelized section of the workflow.

[5] Legend shown in Table 1, page 5.

[6] https://www.openmp.org

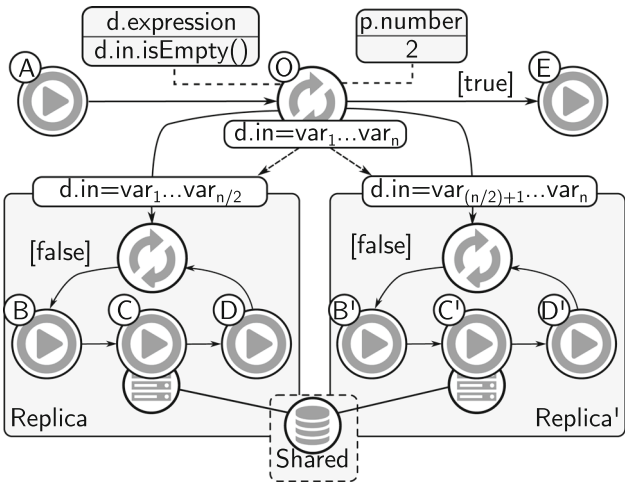


Fig. 9 Twofold parallelization of the loop from Fig. 7

To provide an example of the parallelization process, Fig. 9 shows the loop from Fig. 7 in an active state.[7] However, this time the loop is tagged as parallel with the p.number attribute set to two. As visualized, the start trigger of the loop results in the creation of two nested loops. One contains the tasks (B, C, D) and one contains the replicated versions (B′, C′, D′). In this case, the items to be processed are separated between the nested loops, so that the Replica loop iterates over the first half of the items ($var_1...var_{n/2}$) and the Replica′ iterates over the second half ($var_{(n/2)+1}...var_n$). As soon as the nested loops reach the finished state, the original loop (O) is notified and also transferred to the finished state triggering the reevaluation of the d.expression attribute. As meanwhile each item has been processed and the input is now empty the expression evaluates to true allowing task E to be processed next. This in turn results in the task E being scheduled again. In the following, the SWFMS built around the runtime model is described, including a description of how the required infrastructure of nested loops are derived.

4.3 Model-driven workflow management system

To show the feasibility and expressiveness of a workflow runtime model, we developed a SWFMS that forms a MAPE-K loop. As shown in Fig. 10, the design time Workflow Model serves as input for the Workflow Engine. In this process, the design time model, incorporating all the individual infrastructure requirements, is used to set the high-level goal of the SWFMS coupled with the aim to bring each modeled task into a finished or skipped state. To reach this goal, the Workflow Engine utilizes the Runtime Model as a knowledge pool providing access to the workflow and infrastructure state. Furthermore, the engine uses the Runtime Model as access

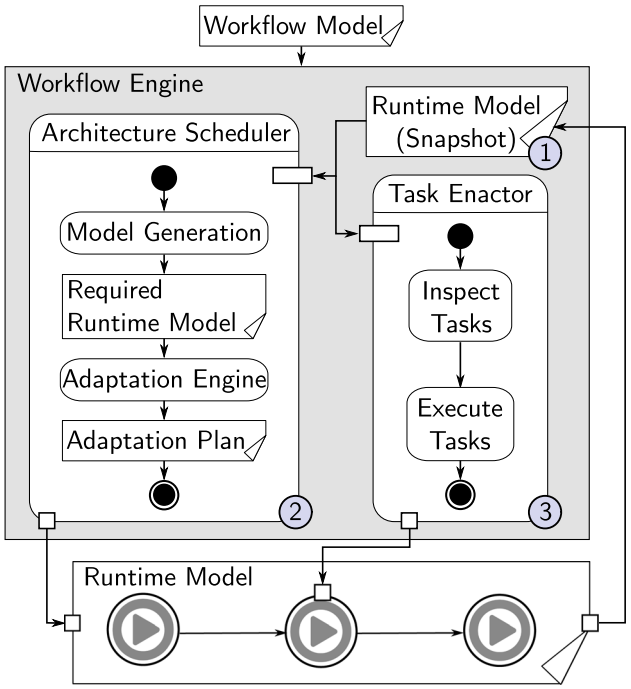


Fig. 10 Overview of the workflow management system

point to enact the workflow and deploy the underlying infrastructure. To initialize the workflow, we empty the Runtime Model so that the model does not contain any resources. Once started, the engine creates a Runtime Model (Snapshot) ① which is passed to its inner components, i.e., Architecture Scheduler ② and Task Enactor ③. In the following, these components are described in more detail.

4.3.1 Architecture scheduler

The Architecture Scheduler ②, shown in Fig. 10, schedules and deploys the infrastructure required by the individual tasks. In each self-adaptive control loop cycle, the scheduler combines the information of the design time Workflow Model and Runtime Model. Within this Model Generation, a new Required Runtime Model state is created describing the infrastructure state required next. The generated model then serves as input for an Adaptation Engine, which triggers a model transformation generating an Adaptation Plan from the Runtime Model and the Required Runtime Model. This plan describes a sequence of imperative requests to be performed in order to reach the new desired state. Finally, the adjusted runtime model is synchronized with the infrastructure due to their causal connection. Depending on the new desired state this plan manages not only infrastructure and platform elements, but also entities from the workflow layer. In the following, the processes and transformation rules of the Model Generation and Adaptation Engine are described in more detail.

[7] Legend shown in Table 1, page 5.

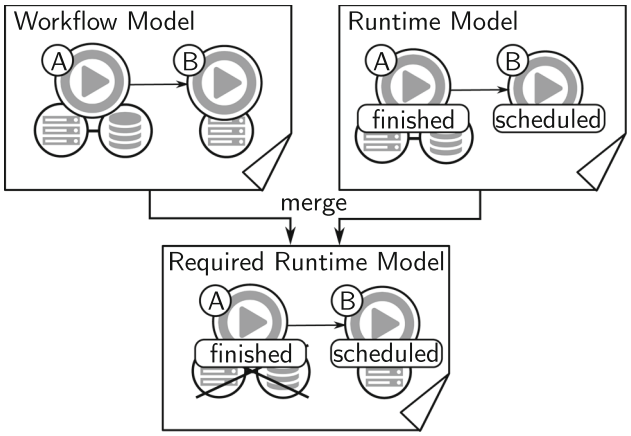


Fig. 11 Required runtime model generation example

Required Runtime Model Generation The generation of the required runtime model forms the core of the analysis and planning phase from the scheduling process. It is responsible to create a runtime representation of the system currently required by the workflow by combining the design time with the runtime information. Hereby, the scientist's workflow is taken as a basis. Then, as shown in Fig. 11, we merge the information of the Workflow Model with the current state of the Runtime Model.[8] Thus, the generation process has access to the workflow, as well as the complete infrastructure required by the individual tasks. Next, resources that are currently not required are removed from the model resulting in the Required Runtime Model.

To identify the resources to remove, the actual state of each task has to be known. Therefore, we first propagate the state of the tasks from the Runtime Model to the Required Runtime Model. Thereafter, we separate the tasks into categories based on their states and identify whether a task is ready for its execution, i.e., each task having all its preceding tasks in a finished or skipped state. Based on the classification, we form a Required Runtime Model that only contains the infrastructure of each task that is either active or ready for execution. As a result, we remove the infrastructure of each finished or scheduled tasks.

In the example shown in Fig. 11, the design time Workflow Model describes two tasks to be executed. This model is copied and serves as basis for the generation of the Required Runtime Model. First, we transfer the states of the Runtime Model, i.e., the finished state for task A and the scheduled state for task B. Then, we identify task B as ready for execution as all previous task are finished. This in turn allows to remove the infrastructure resources of task A leaving the infrastructure requirements of task B in the model. Finally, the merging process results in a Required Runtime Model that only contains the infrastructure for the tasks to be executed next, i.e., in this case one compute node.

When a parallelized loop is executed, multiple replicated tasks exist within the runtime model. Thus, compared to the standard merging process, additional infrastructure has to be added to the model to satisfy the infrastructure requirements of new spawned tasks. We identify these tasks over the replica tag allowing the merging process to add the underlying infrastructure for these tasks. As a first step, we duplicate the infrastructure of the modeled original task. The duplicated resources are then attached to the loop replica and its task. Hereby, everything is duplicated except resources tagged as shared, including their sub-resources. Even though shared resources are not replicated, the edges towards these nodes are reproduced granting access to the shared node in each replica group. Each duplicated infrastructure resource is also tagged as a replica. As this process is loosely coupled with the system, it can be easily enhanced to utilize further information, e.g., the estimated execution time of a task to allow for early deployments. It should be noted that the resulting required runtime model can serve as input for further transformations, e.g., for platform specific adjustments.

Adaptation Engine To bring the infrastructure to the state described in the required runtime model, we reutilized a model-driven adaptation engine already described in previous work [67]. In general, this engine compares the declarative states of the runtime and required runtime model with each other to derive a set of imperative steps to adapt the runtime model to the desired state. Hereby, it compares the runtime model to the required runtime model identifying new, missing, adjusted and already existing elements. Based on this match, the engine identifies elements to delete, create and adjust. To determine the order of adaptive actions to be taken, a *provisioning order graph* [68] is generated from the required runtime state. This graph describes the dependencies between elements to be created within the runtime model. The resulting graph allows to describe, e.g., that a VM and network need to be created before they can be linked. From this graph, the already existing elements are removed and an activity diagram is generated forming the Adaptation Plan shown in Fig. 11. Finally, this plan is executed adjusting the runtime model and thus the causally connected infrastructure.

As the workflow and infrastructure layer are both managed over the runtime model, this engine is responsible to not only provision the infrastructure, but also to initially create the workflow elements within the runtime model. On the first cycle of the workflow execution process, no task is currently scheduled. As a result, the engine propagates the state of the workflow elements directly to the runtime model. Therefore, at the beginning of a workflow's execution, the runtime model solely consists of the designed workflow elements leading to a creation of the required infrastructure next.

[8] Legend shown in Table 1, page 5.

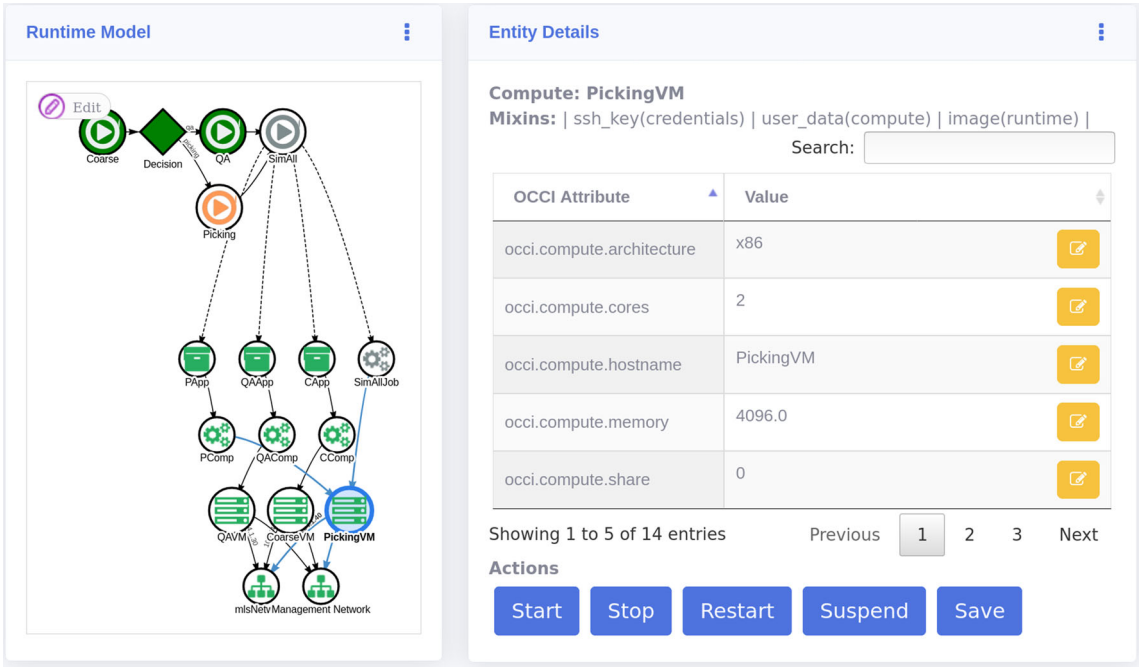


Fig. 12 Screenshot excerpt from the SmartWYRM dashboard

4.3.2 Task enactor

The Task Enactor ③, shown in Fig. 10, checks for tasks being ready for execution and enacts them accordingly. Compared to the scheduler, the enactor directly operates on the task elements by triggering their start action. Furthermore, depending on the configuration of the engine the enactor and scheduler may run in separate cycles. As part of the Task Enactor process, two subsequent actions are performed. First, the enactor performs an Inspect Tasks step investigating a set of tasks ready for execution, which in the Execute Tasks step get triggered.

During the Inspect Tasks step, we filter for tasks that have all their previous tasks either in a finished or skipped state. Furthermore, another requirement is that all their platform dependencies have already been fulfilled. We define the platform dependencies as fulfilled, when the application or component connected by a platform dependency is in an active state indicating that it is successfully deployed and running.

In the Execute Tasks step, the enactor triggers the start action for each task ready to run. This brings the task into an active state, which also deploys and executes its executable component. When the task is a decision or a loop, additionally the decision-making process is performed. If no data gathering process is modeled at design time, the decision-making information can be manually filled at runtime. The decision node remains in an active state until its decision input has been filled allowing scientists to decide which control flow to follow.

5 Implementation: SmartWYRM

To evaluate the runtime model concept, we developed the *Smart Workflows through dYnamic Runtime Models* (SmartWYRM)[9] framework with the goal to dynamically utilize arbitrary infrastructure for individual tasks within a workflow. SmartWYRM is implemented as a Java web application providing a dashboard and visualization of the runtime model that can be accessed and changed at runtime. A screenshot of this dashboard is shown in Fig. 12 highlighting the options of a selected compute node. While selected, the attributes of the element can be changed allowing, e.g., to adjust the amount of compute cores of the selected machine. The interaction capabilities of the workflow are hereby determined over designed and registered OCCI extensions. In addition to changing attributes, the interaction capabilities include standardized OCCI actions, e.g., to start and stop a VM, as well as actions of custom extensions like stopping a task to interrupt the workflow. Especially, the extension mechanism of OCCI allows scientists to design desirable actions to quickly interact with a workflow at runtime or define specific routines to handle intermediate results. Additionally, the server records and stores information about the

[9] https://gitlab.gwdg.de/rwm/smartwyrm

workflow execution as part of a job history. This job history visualizes the time required for the workflow, scheduling and task execution while tracking the individual requests and runtime model states reached during workflow execution. As shown in Fig. 13, the SmartWYRM ① framework represents the implementation of the model-driven Workflow Engine processing a Workflow Model designed by the user. Hereby, the user requires fundamental knowledge about the OCCI standard and the cloud domain to design and interact with the model. The Workflow Model and Runtime Model are both an instance of an *Eclipse Modeling Framework* (EMF) metamodel [69] allowing us to utilize the Epsilon language family [70] for the presented transformations and model generations.

To implement our workflow runtime model, we chose to extend the OCCI ② cloud standard due to several reasons. First OCCI specifies a standardized, as well as an extensible data model [11] for which a precise and formal EMF OCCI Metamodel already exists [71]. Second, the standard introduces a uniform OCCI Interface [72] which the Workflow Engine can use to create the infrastructure and perform the workflow, as well as an interface to manage model elements. Third, the OCCI data model builds around resources and links, similar to nodes and edges used by DAGs, allowing us to create an OCCI Extension to support the workflow elements discussed in Sect. 4. To generate this extension, we used OCCI-Studio,[10] which is part of the OCCIWare toolchain [73] that allows to design and generate OCCI extensions. Due to the high level of abstraction introduced by this toolchain, users can extend the OCCI metamodel and therefore improve the capabilities of their designed Workflow Model. In order to extend the model with self-developed frameworks, knowledge about configuration management languages, like ansible, is required to enable an automated deployment.

Part of this toolchain is the OCCIWare Runtime ③ server application[11] in which one Connector for each OCCI extension can be registered. A Connector describes how incoming OCCI requests are processed. In case of Infrastructure resources, it describes how incoming OCCI requests are translated to requests conforming to the interface of the chosen cloud provider. Additionally, this server is responsible to maintain the Runtime Model providing us with the current state of the workflow and deployed infrastructural resources. Depending on the implemented effector different model-driven techniques can be utilized. For example, the platform effector [65] uses Acceleo [74], a code generation tool, to generate variable files from the runtime model to be used within configuration management scripts.

[10] https://github.com/occiware/OCCI-Studio

[11] https://github.com/occiware/MartServer

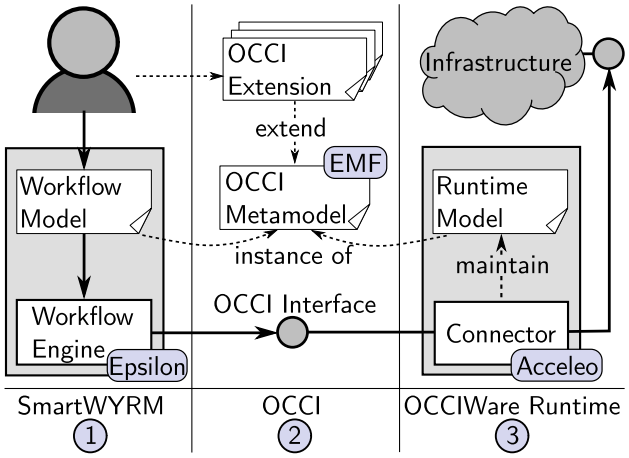


Fig. 13 SmartWYRM in the OCCI ecosystem

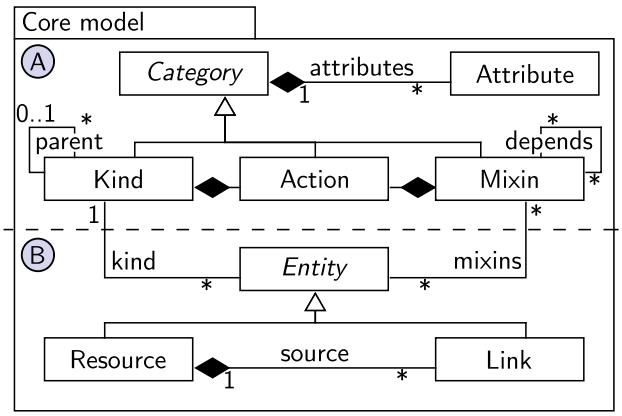


Fig. 14 OCCI Core data structure (adapted from [11])

In the following, we provide a brief introduction into the OCCI standard, its ecosystem and the implementation of the workflow runtime model.

5.1 Open cloud computing interface

The OCCI Core Model [11], shown in Fig. 14, introduces a set of core elements meant to be extended in order to manage cloud resources. This extendable core consists of elements inheriting from the abstract type *Category* forming a classification and identification mechanism Ⓐ as well as core base types Ⓑ inheriting from the abstract *Entity* type.

Each *Entity* is of one specific Kind. A Kind defines a set of Attributes to be filled with values by the *Entity* and Actions that can be performed on it. *Entity* is an abstract core type and needs to be instantiated either as a Resource or a Link together describing cloud deployments. A Resource resembles the notion of a node, while Links resemble the notion of an edge within a DAG. A VM, e.g., is described by a Resource of Kind Compute. This Kind defines attributes such as the amount of memory, as well as actions to start and stop the machine.

When triggered, an Action affects the state of the resource as described by a FSM. In addition to a Kind, each *Entity* may have multiple attached Mixins. These describe capabilities to be added to the *Entity* at runtime and resemble the notion of a tag that can be added to nodes or edges. The capabilities added by a Mixin are again defined over a set of Attribute and Action instances that can be performed on an *Entity*.

5.2 Runtime model implementation

The implementation of the runtime model is based on the formal OCCI EMF metamodel defined by Merle et al. [71] which later got enhanced in [73]. This metamodel builds the basis for a model-driven toolchain to design, create and validate OCCI models, extensions and interfaces. Using this toolchain, the standardized OCCI extensions, e.g., the infrastructure [66] and platform [63] extensions, got instantiated. Furthermore, multiple non-standardized extensions are developed using this toolchain, e.g., to support container [75] or configuration management [65] using OCCI. To handle infrastructure elements, we used the OCCI infrastructure extension [66], while we manage the platform layer via the *Model-Driven Configuration Management of Cloud Applications with OCCI* (MoDMaCAO) framework [65]. MoDMaCAO enhances the original platform extension [63] and allows to model applications to be deployed via configuration management scripts, like ansible, on top of provisioned infrastructure. To ensure a connection to the VMs to be configured, a management network is required. Therefore, we add a management network by default to the required runtime model via an additional model transformation. For the decision-making processes, we used the monitoring extension [64]. This extension allows to manage sensor and reflect monitoring results within the runtime model. Finally, for the workflow layer, we implemented the workflow extension [12, 13] and enhanced it with the loop and parallelization elements concept. The behavior of each individual element ensuring the causal connection to the system is implemented within connectors for which skeletons can be generated from modeled OCCI extensions. It should be noted that the OCCIWare toolchain can be used to easily create domain-specific extensions. These allow to define specific deployment components providing them access to user-defined attributes that can be easily adjusted.

Overall, the connectors as well as the OCCI extensions serve as plugins for the OCCIWare runtime server. This server accepts incoming OCCI requests, handles the management of the OCCI runtime model, and triggers the behavior defined within the connector for the individual OCCI element. For example, when creating a VM, an OCCI request is sent to the OCCIWare runtime server identifying the requested element as compute node. Then, within the connector of the infrastructure extension, the behavior is defined to provision the modeled VM within a cloud environment. Similarly, within these connectors, we observe the target environment to get notified on changes allowing us to update the runtime model ensuring a causal connection between the model and the system.

In addition to the described connectors, we use a set of dummy connectors to simulate the behavior of the runtime model. These connectors implement the default behavior of the runtime model, i.e., the management of model elements, the execution of actions, and the transition of states. Moreover, simulation behavior may be added, e.g., to add monitoring results that can be observed by modeled sensors. Overall, these dummy connectors allow to simulate the workflow on a model-based level supporting the development of scientific workflows, as their behavior can be observed without an actual execution in the cloud.

6 Case studies

To show the feasibility of our approach, we created and executed models for scientific workflows from three domains using the SmartWYRM framework. For the case studies, a private Openstack cloud is used, as well as the OCCIWare runtime server managing the runtime model. Additionally, we executed each of these case studies using the dummy connectors simulating the workflow behavior as described in the previous section. The first study (Sect. 6.1) shows the integration of a common big data computation framework, as well as dataflows between tasks hosted on distributed machines. The second study (Sect. 6.2) covers the decision-making process within a dynamic simulation scenario. Moreover, this case demonstrates the utilization of other infrastructure resources within the workflow, such as the provisioning of specific networks. The third case study (Sect. 6.3) describes a software repository mining workflow showing the utilization of loops and parallelization. At the end of each case study section, the results of each study are described focusing on design and runtime observations.

For each case study, we provide a brief introduction to the domain, as well as a detailed description and visualization of the workflow runtime model using the icons introduced in Table 1. We focus on how the workflow model changes and visualize the workflow execution as a sequence of runtime model states. These states represent the results of the required runtime generation and are reached via the transformation and execution of the adaptation plan. It should be noted that greyed-out elements within the visualization of the model describe elements that are currently not provisioned. We omit the visualization of the management network, as it is required by the MoDMaCAO framework to configure the provisioned VMs.

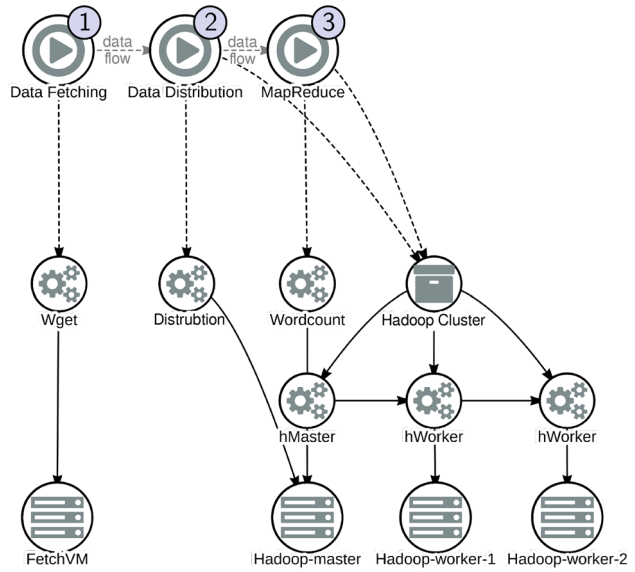


Fig. 15 On demand big data workflow

In addition to the description of the runtime model states during the workflow execution, we measure the *scheduling time*, as well as the duration of the *task execution*, *required runtime model generation* and the *overall workflow*. Each case study is executed five times with the gathered values being used to calculate and visualize the 95% confidence interval and mean for each of these values. We use the observed results to relate the execution times of the different parts of our approach and discuss potential overheads.

6.1 On demand big data framework

The need to analyze large amounts of data has led to a tremendous amount of distributed analytic frameworks, such as apache hadoop [76] and spark [77]. These frameworks are designed to process large data sets using, e.g., the MapReduce programming model [78], for which a common example is to create a *bag of words*. In a bag of words, the number of occurrences of each word is counted from a data set. In this case study, we demonstrate a workflow in which a hadoop cluster is deployed to create a bag of words based on text files gathered by the workflow beforehand. While only a few or a single resource is required to gather the data, a large computation cluster is needed to analyze the data.

6.1.1 Workflow model

The workflow is shown in Fig. 15 and consists of three tasks: Data Fetching ①, Data Distribution ② and the MapReduce ③ task.[12]

The Data Fetching ① task requires only a single VM (FetchVM) as it is only responsible to download a set of text files to be analyzed via the Wget executable component. Thereafter, the fetched data are transferred from the FetchVM hosting the Wget executable component to the VM hosting the Distribution executable, which is modeled by a dataflow connecting the Data Fetching and Data Distribution task.

The Data Distribution ② task takes the downloaded text files and stores them in the *Hadoop Distributed File System* (HDFS) [79], a distributed file system on which the hadoop computation cluster resides. To fulfill this task, the deployment of the hadoop cluster is already required, as shown by the modeled platform dependency to the Hadoop Cluster application. This cluster consists of three VMs including one master and two worker nodes. Within this application, the hMaster application component is connected to the hWorker application components using an *execution dependency* which is a specialization of a ComponentLink [65]. This link allows to describe the order in which the individual components should be deployed and started once the start action on the Hadoop Cluster application is triggered. Furthermore, this link provides access to attributes set within the connected entity, as well as its surrounding entities. This in turn allows to adjust the deployment according to the current runtime information. In this case, this connection is used to register the hostnames of each VM being part of the Hadoop Cluster.

The MapReduce ③ represents the last task within the workflow and is linked to the Wordcount executable, which is hosted on the Hadoop-master compute node. Also this task requires the Hadoop Cluster to be deployed which provides the computing utilities and stores the text files needed to perform the task.

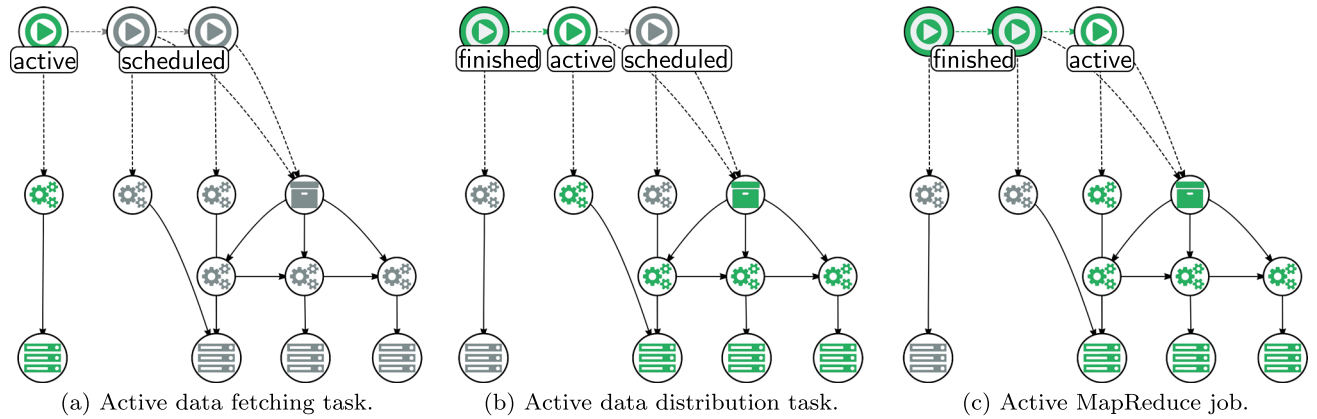
In this workflow model, the task dependencies between the individual tasks represent a dataflow. The information about a tasks input and output data resides within this link. Depending on the data locality, the dataflow itself can be triggered transferring the data from one machine to another. In this case both machines need to be active and running before the data can be transferred. It should be noted that due to the availability of the infrastructure layer different communication channels can be chosen to transmit the data like the management network of the hadoop cluster. To choose the mentioned network, the network's id can be configured within an OCCI Mixin that can be attached to the dataflow link.

6.1.2 Workflow execution

Figure 16 shows three major states of the runtime model during the execution of the big data workflow shown in Fig. 15.[13]

[12] Legend shown in Table 1, page 5.

[13] Legend shown in Table 1, page 5.



(a) Active data fetching task. (b) Active data distribution task. (c) Active MapReduce job.

Fig. 16 Workflow runtime model progression showing the dynamic management of resources for the deployment and operation of a big data framework

The individual time steps represent the results of the generated and deployed required runtime model with green nodes representing active tasks or deployed entities and gray nodes undeployed resources or scheduled tasks. Finally, tasks with a green background and a gray icon are in the finished state.

In the first time step (Fig. 16 (a)), the Data Fetching task in its active state is shown. This task is the first one triggered by the workflow engine, as it has no previous tasks. Therefore, the generation of the required runtime model results in a model that contains only the infrastructure of the Data Fetching task which in this case is a single VM having the wget application already pre-installed by the operating system. In the same cycle, once the VM is started, the Data Fetching task is triggered resulting in the deployment, configuration and start of its executable component fetching a set of text files. In this case, a list of text books is downloaded from an internet archive. Once all the text files are downloaded, the Data Fetching task reaches the state finished and the fetched data is made available for the Data Distribution task. Therefore, a dataflow is triggered transferring the data from the VM hosting the executable of the Data Fetching task to the VM hosting the Data Distribution task. Therefore, both host VMs get started by generating a corresponding required runtime model.

After the data have been transferred, the dataflow reached a finished state leading to the second time step (Fig. 16 (b)), showing the active state of the Data Distribution task. As this task requires the deployment of the Hadoop Cluster application, which is modeled by a platform dependency. This results in the second and third VM being provisioned used as worker nodes in the cluster. Furthermore, in this time step the VM used to fetch the data gets removed from the runtime model as it is not required anymore. Due to the causal connection of the runtime model to the infrastructure this consequently leads to the release of the VM in the cloud environment. As soon as the cluster application has been deployed, the second task is triggered and the fetched data is loaded into the HDFS.

Finally, in the last time step (Fig. 16 (c)) the active MapReduce task is shown. This task can now be triggered, as the workflow and infrastructure requirements for the task are met. Consequently, the task enactor triggers the execution of the hadoop job computing the bag of words. Once the job is completed, each task has reached the finished state, stopping the workflow engine. It should be noted that in order to release excess resources a cleanup task may be modeled which, e.g., stores the result of the Wordcount job in a database and releases the deployed Hadoop Cluster application. Otherwise, the infrastructure of the last task remains provisioned as the engine considers the workflow to be finished once all tasks have been completed.

6.1.3 Results and observations

The case study demonstrates the utilization of already existing frameworks. We showed that the infrastructure can be changed by generating required runtime model states for each point in time throughout the workflow execution. Furthermore, the case study showed the feasibility of a workflow runtime model to utilize dataflows to transfer data between subsequent tasks.

At design time, the complexity of the workflow model differs for the individual layers. Modeling the workflow layer itself was simple as it only consists of three subsequent tasks having clearly defined platform requirements. Also, the associated executable components require no effort in their creation as they contain simple scripts triggering other programs. Still, they had to be adjusted to utilize the information contained within the dataflow correctly. The complexity of the workflow lies within the application components, as the configuration management scripts used for the deployment of the hadoop cluster are written in such a manner that addi-

tional workers within the cluster can be simply attached and released by utilizing the information in the runtime model. Nevertheless, the creation of the scripts represents a one-time effort for future workflows requiring the deployment of a hadoop cluster of different sizes. Therefore, to scale the amount of workers, compute nodes that host a worker component can be simply added to the workflow model. Moreover, due to the access to the infrastructure layer further tweaks are possible, e.g., to save resources and reuse the hadoop master compute node also for the data fetching task.

At runtime, the case study showed that the overall time required to execute the workflow can be separated into the deployment time and the time in which tasks are actually performed. While the deployment time is most of the time fixed, the time to execute the tasks may differ based on their given input. Independent of whether a task is currently executing or infrastructure being deployed, the visualization of the runtime model provided insight about, e.g., the current state of an application and its components.

The timing measurements are visualized in Fig. 17 with the 95% confidence interval and mean of the overall workflow, task execution, scheduling and model generation duration. Overall, the mean of the complete Workflow execution in our experiment is 580 s with its variance caused by the Task execution. While the mean of the Task duration is 360 s, the execution time varies, due to the Data Fetching task, as in this step multiple files are downloaded with the time depending largely on the download mirror speed. The Scheduling of the architecture on the other hand requires 210 s in the mean. The low variance of the provisioning and deployment process mainly occurs due to the time required to spawn up the individual VMs. The deployment time of the hadoop application is rather constant, as no external download mirror is utilized, which allows uploading the binaries directly from the OCCI server to the compute nodes. Overall, for this workflow the largest portion in the provisioning and deployment process is used for the provisioning and deployment of the hadoop cluster application ranging between 161 and 176 s. Finally, the execution of all required Model Generation procedures during the workflow execution took 0.7 s in the mean. The time requirements for each cycle in the workflow execution took less than 0.3 s per iteration.

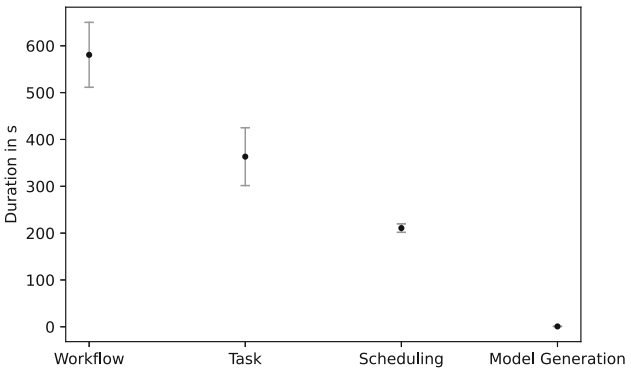


Fig. 17 Hadoop workflow 95% confidence interval plot for the gathered timing measurements in seconds

6.2 Dynamic simulation

The utilization of distributed environments for simulations can be of great benefit as the amount of computing resources can be scaled on demand and thus help to reduce costs and energy consumption. In this scenario, we present a workflow for a multi-level simulation in which parts of the simulation can be performed on different levels of detail [80]. Within the case study, we support these different levels of detail by adding and removing infrastructural resources required at runtime. The simulation used within the following case study describes the supply chain of a factory that is simulated on a coarse level with detailed simulations that can be added to describe the order picking and quality assurance process of the factory in more detail. The workflow and the case study focus on the decision-making capabilities as different simulation levels can be triggered on demand. This case study builds upon the one introduced by Erbel et al. [13] and extends it by a more detailed description of the workflow model and its execution.

6.2.1 Workflow model

The workflow of this case study is shown in Fig. 18.[14] This model comprises the task sequence as well as the required infrastructure for each task, i.e., Coarse ①, Decision ②, Picking ③, QA ④ and SimAll ⑤.

It should be noted that for all simulations, including the detailed simulations, the deployment of the CApp ⑥ is required. Therefore, each task is connected via a platform dependency to the application. We omitted these dependencies from the figure for clarity.

The Coarse ① task represents the standard execution of the simulation without much detail, while subsequent tasks add further details to the simulation. Each task requires the deployment of a specific application created by the scientist to perform the simulation, e.g., the coarse simulation requires the CApp. In addition, each task has a dedicated executable, which in case of the coarse simulation is the CJob component. These jobs trigger the deployed simulation application with parameters defined by the scientist, e.g., the amount of simulation steps to be performed. Each part of the simulation is hosted on a separate VM which are connected over a specific network (mlsNetwork). This network is configured to handle the network traffic of the simulation and gets automatically provisioned at runtime.

[14] Legend shown in Table 1, page 5.

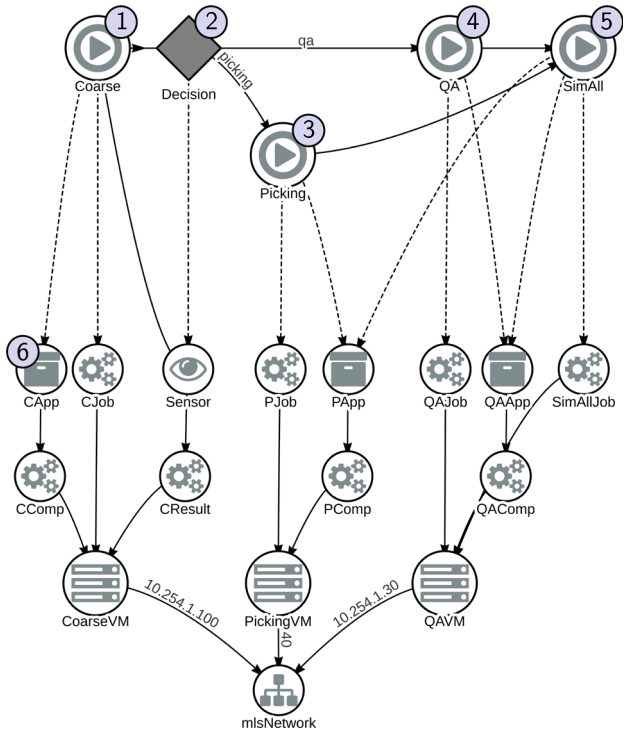


Fig. 18 Multi-level-simulation workflow

After the first simulation time steps have finished, the Decision ② node decides which detailed simulation to add. Therefore, the results of the coarse simulation are observed using the Sensor. This Sensor consists of the component CResult, which investigates the output of the coarse simulation and evaluates which kind of detail should be added to the simulation. The result itself is reflected within the link pointing from the Sensor to the Coarse task. As the Sensor represents the executable of the Decision node, this information is additionally used as decision input. Based on the decision input and the control flow guards (picking and qa) one of the detailed simulations is added to the coarse simulation.

The Picking ③ task adds detail of the order picking process of the packages within the factory. Therefore, it requires the deployment of the PApp simulation application and its PJob executable. Both are hosted on the PickingVM which is connected to the mlsNetwork.

The QA ④ task adds a detailed simulation of the quality assurance process. Similar to the picking simulation, the deployment of the simulation application and the executable are required, namely QAApp and QAJob. Both are hosted on a dedicated QAVM connected to the mlsNetwork.

Finally, the SimAll ⑤ task adds all levels of detail to the simulation. Therefore, the task requires the deployment of each simulation application, i.e., CApp, PApp and QAApp, including the compute nodes they are are hosted on, i.e., CoarseVM, Picking VM and QAVM. Additionally, the task is connected to its own executable component SimAllJob triggering the simulation on the highest level of detail.

6.2.2 Workflow execution

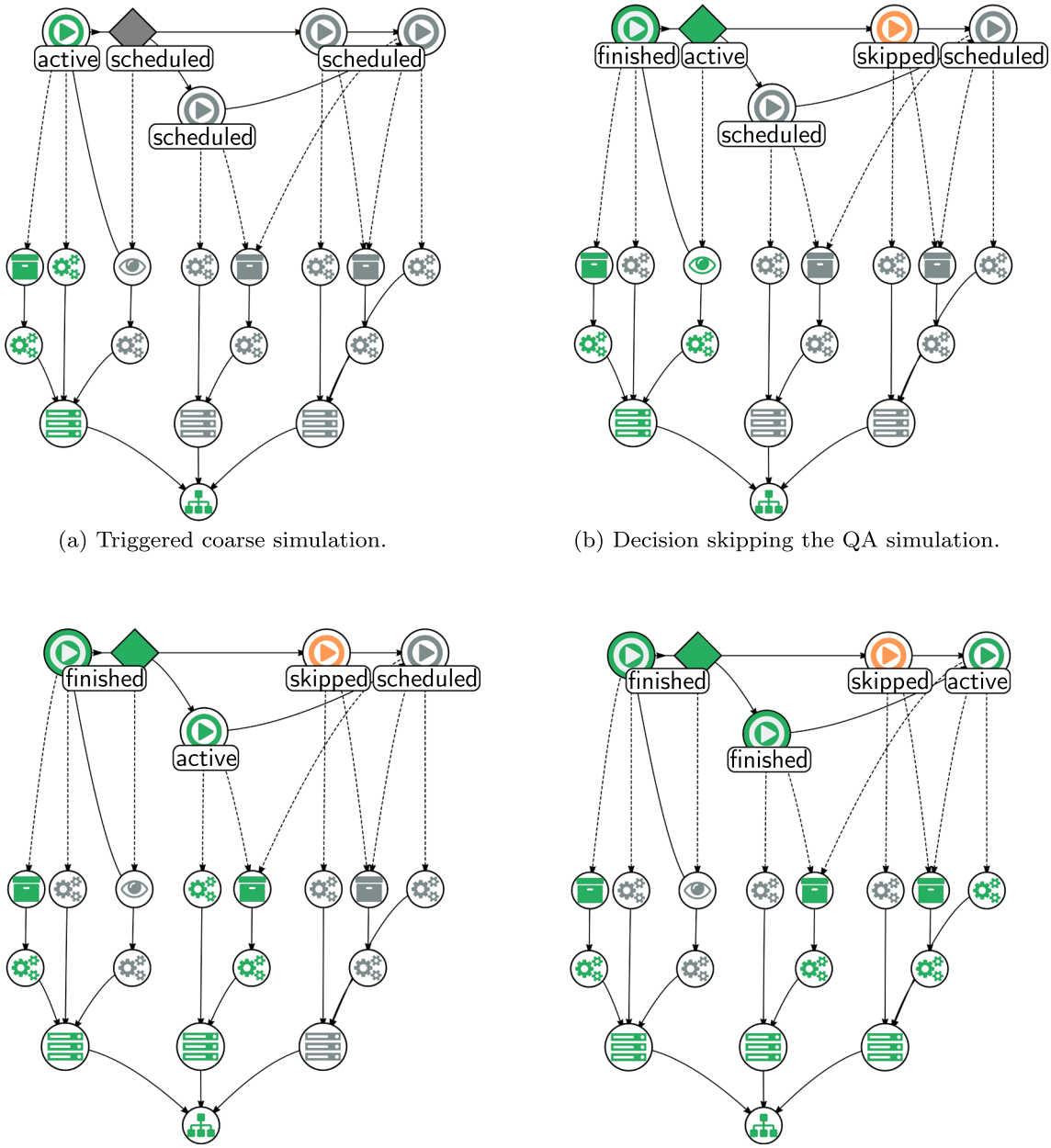
When triggering the execution of the simulation workflow, the Coarse task is marked as ready for execution as it has no preceding tasks. Therefore, its required architecture is the only one contained within the first required runtime state (Fig. 19 (a)).[15] This comprises the VM for the coarse simulation, as well as its application and simulation executable. After the adaptation plan has been executed and the VM and the application are up and running, the task enactor triggers the execution of the Coarse task and thus the executable starting the coarse simulation resulting in the task being active.

As soon as the coarse simulation is finished the workflow engine identifies that the Decision task has to be performed next (Fig. 19 (b)). The task enactor triggers the execution of the Decision bringing it to the active state. Activating the Decision node results in the execution of its Sensor observing the outcome of the coarse simulation. Based on the monitored information, the decision input of the Decision task is filled and compared against the control flow guard of each task dependency connected to it. It should be noted that the Sensor already identifies which detailed simulation should be added resulting in the decision input being either "qa", for the quality assurance simulation, or "picking", for the order picking simulation. In the depicted time step, the decision input is set to picking. This results in the QA task transferred to the skipped state while leaving the Picking task in the scheduled state and ready to be executed next. It should be noted that the same runtime states can be observed when manually adjusting the decision input and control flow.

Independent of how the decision input of the Decision node has been filled, it now has reached the finished state and the task to be executed next is in a scheduled state. Now the scheduling process knows which infrastructure is required next (Fig. 19 (c)). In this case only the infrastructure for the Picking task is required. Thus, the Sensor is deprovisioned and a new VM hosting the application and the job for the order picking simulation is deployed. After the deployment, the Picking task is executed and now active.

As soon as the picking simulation is finished, its corresponding task reaches the finished state (Fig. 19 (d)). Thus, all workflow requirements to execute the final SimAll task are fulfilled as each previous task is either in a finished or skipped state. Therefore, the engine provisions and deploys the required resources, i.e., the VMs and application of each individual simulation level, as well as a single executable to trigger the simulation. Finally, the task enactor triggers the execution of the overall simulation job which is now active.

[15] Legend shown in Table 1, page 5.



(a) Triggered coarse simulation.

(b) Decision skipping the QA simulation.

(c) Activation of the picking simulation.

(d) Activation of all detailed simulations.

Fig. 19 Dynamic simulation workflow execution. Green nodes represent active or deployed entities, gray nodes undeployed resources or scheduled tasks, orange tasks are skipped, and tasks with a green background and a gray icon are finished

Once the simulation task is finished the workflow is completed. Again, a cleanup task can be modeled to store the results of the simulation and release excess resources.

6.2.3 Results and observations

The case study demonstrated the decision-making, monitoring and interaction capabilities of the workflow runtime model. It showed that the results of a task can be reflected in the model and used to choose a specific control flow. Furthermore, we showed that over a direct interaction with the model the control flow can be changed by a human-in-the-loop.

At design time most of the presented workflow fits into the general schema of our approach. In this study, the complexity of the model lies within the configuration of a dedicated network to be used by the simulation. The setup for this network required manual effort specific for this use case. Furthermore, as each task within the workflow requires the application of the coarse simulation to be deployed, a lot of platform depen-

dencies had to be modeled cluttering the overall visualization of the workflow at design time.

At runtime, the high amount of cluttered platform dependencies are not present in the visualization of the workflow model. This happens as only a small subset of all platform dependencies exist at each point in time within the runtime model providing a good overview of the infrastructure and the workflow. The reflection of the runtime information within the model was easy to integrate through the modeled sensor, even though it had to be manually configured for this use case. Changing parameters in the model itself only required a single request to OCCI's uniform interface allowing to adjust the control flow as only the decision input had to be changed. In this regard, the automatic decision-making can be disabled by removing the sensor from the model. In return the decision-making has to be done at runtime by the scientist as a human-in-the-loop. To further investigate the runtime model capabilities, we wrapped the simulation tasks in a loop. This loop lets the simulation run until the scientist stops it or a specific amount of iterations has been reached. This allows choosing smaller simulation time steps and a more detailed management of detailed simulations to be added. The described simulation workflow case study represents one cycle within the looped model. It should be noted that the decision input remains at the last decision made by the scientist during each iteration. Therefore, manual input was only required once a different detailed simulation is desired.

The timing measurements for the automated decision-making workflow is shown in Fig. 20. The figure shows that the overall Workflow execution in the mean took 344 s. For this workflow, the variance for the Task execution is constant with a mean duration of 50 s. The Scheduling on the other hand took 284 s in the mean. Finally, the execution of all required model generations during the workflow execution took 1.8 s in the mean. Compared to the other duration, the time requirements for the required runtime Model Generation do not greatly impact the overall workflow execution. Here, the Scheduling of the infrastructure takes the most of the workflow execution time, as the VMs and applications need to be deployed for the individual simulation parts. The Task execution time on the other hand only requires 50 s, revealing that the simulation only makes sense with a sufficiently large number of simulation steps.

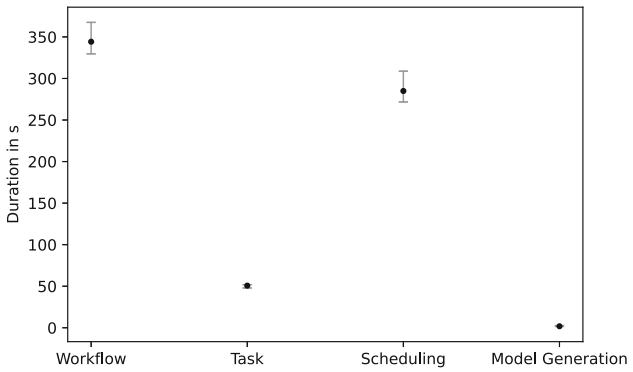


Fig. 20 Simulation workflow 95% confidence interval plot for the gathered timing measurements in seconds

6.3 Software repository mining

In the domain of software repository mining, project information stored in a *Version Control System* (VCS), issue tracking system and/or mailing lists is analyzed to draw conclusions about specific phenomena. During this process, a dynamic amount of computing resources is required that heavily depends on the kind of task within the scientific workflow. In this case study, we investigate the mining and analytic process that is part of the SmartSHARK platform [81]. This platform is used to perform empirical studies about software projects, e.g., to gain insight about the trends of static analysis warnings in open-source projects [82]. While the SmartSHARK platform consists of multiple plugins, we focus on the ones used for the mining process. The mining process is composed of three steps. First, metadata is extracted from a repository and stored within a database. Next, each commit of a software project is checked out to analyze its metrics. The result is then stored in the same database. As a final step, redundancies in the database are cleaned up to reduce the amount of required storage. While the first and the last step of this process require only a few computing resources, the second step may utilize as many computing resources as available. Especially as this step can be easily parallelized. Also the required size of computing resources in this step largely depend on the size and amount of files in the software project to analyze as well as the number of commits.

Within this case study, we utilized OCCIWare and the MoDMaCAO framework to model and generate a domain specific extension of the SmartSHARK platform. This extension consists of specialized component elements representing the individual SmartSHARK plugins and applications. These elements allow a more precise deployment and configuration of the components due to introduced attributes that can be set by the scientist. For example, this extension allows to choose the repository to analyze by exposing a corresponding attribute in the executable component or to set the desired log level in the application components.

Similar to the case studies above we describe the workflow as well as its execution in the upcoming sections. Additionally, we provide a detailed description of the runtime model state when the loop is marked as parallel. Here, we demonstrate how additional infrastructural resources are created at runtime to process the commits of a software project in parallel.

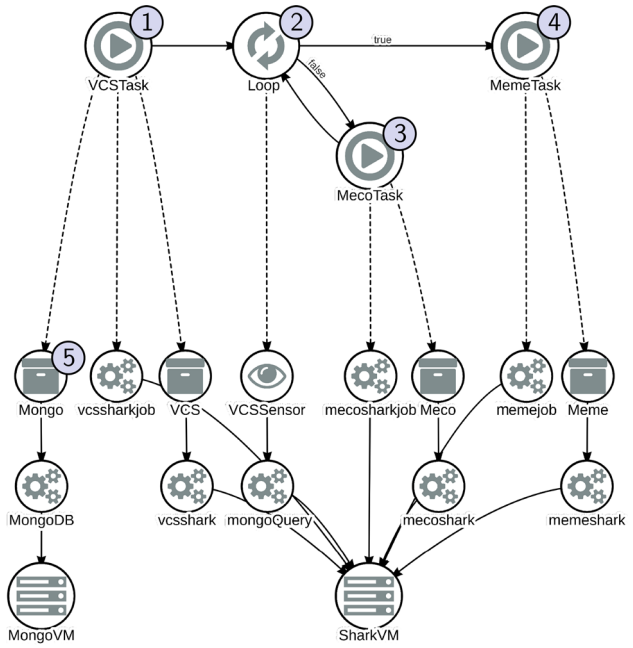


Fig. 21 Software repository mining workflow

6.3.1 Workflow model

The mining workflow, shown in Fig. 21, reflects the three software repository mining steps including a loop iterating over the commits to analyze.[16] These include the VCSTask ①, Loop ②, MecoTask ③ and MemeTask ④. Similar to the simulation use case, each task has an application required to run as well as an executable triggering the application with specific input. Each of the individual tasks requires the MongoDB ⑤ application to run representing the database in which the gathered data are stored. It should be noted that each task is connected over a platform dependency to the MongoDB representing the database that is required throughout the whole workflow. In the workflow figure we omit all but one of these dependencies, connecting the tasks to the database, for clarity.

In this workflow, the VCSTask ① performs the metadata gathering process. To gather the metadata it requires the VCS application to be deployed. Furthermore, the vcssharkjob executable component needs to be deployed and started. When triggered this component starts the gathering process on the repository defined by the scientist.

As each revision of a project needs to be analyzed, the Loop ② iterates over the individual revision hashes queried over the metadata information gathered beforehand. This information is directly retrieved from the MongoDB via the mongoQuery component representing a monitoring instrument of the VCSSensor. As a result of executing the VCSSensor, the Loop is filled with revision hashes to analyze. During each iteration, one revision hash is given to the looped task, leaving the decision input empty once all revisions have been analyzed. In this regard, we check for an empty decision input within the decision expression of the loop. Combined with the guard false attached to the edge leading to the looped task, the loop is active until all items have been processed.

In this workflow, the MecoTask ③ is looped. This task analyzes the metrics of a commit and requires the Meco application to be deployed. The analysis process of this task is triggered over the mecosharkjob executable component connected to it.

Finally, as the last step the MemeTask ④ requires the Meme application to be deployed. This application gets triggered by the memejob executable component shrinking the overall amount of storage required in the database.

Within the workflow model, two VMs reside. The MongoVM hosting the MongoDB and the SharkVM used to process the computation steps analyzing the repository. We reutilize the SharkVM for the individual tasks, as each task requires only a single machine with similar packages being installed. During the execution of the parallel workflow model, this VM is duplicated at runtime processing multiple instances of the MecoTask for different commits.

6.3.2 Workflow execution

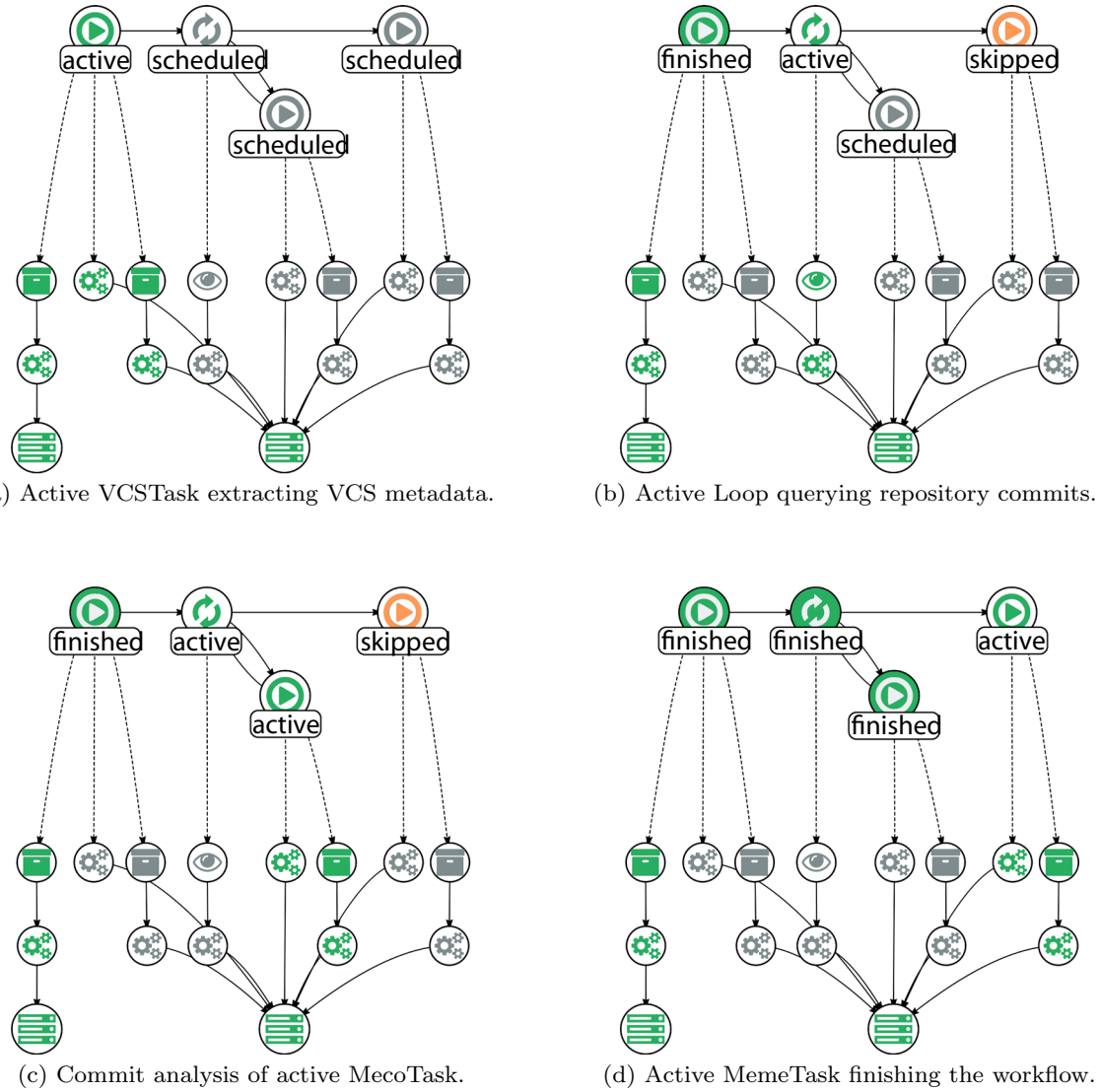
To perform the workflow depicted in Fig. 21, we filled the tasks with information about the repository to mine and started its execution using SmartWYRM. We performed this workflow on the repositories hosting the implementations of our approach which, in relation to other open source projects, represent small projects ranging from 80 to 400 commits. Independent of the chosen project, the execution of the workflow remains the same as depicted in Fig. 22.[17]

At first the VCSTask is executed (Fig. 22a). Due to the platform dependencies from the task to the MongoDB and VCS application, their infrastructure is contained within the required runtime model. This leads to the provisioning of the MongoVM and SharkVM, as well as the deployment of the MongoDB and VCS application including its application components. While the MongoVM persists throughout the workflow execution, the SharkVM is reutilized for the different tasks within the workflow. After the deployment process is finished, the VCSTask is triggered, resulting in it being active and the execution of its vcssharkjob.

Once all the metadata of the repository are gathered, the VCS application is deprovisioned and the Loop is triggered next (Fig. 22b). As the Loop is now in the active state, the VCSSensor attached to the Loop is triggered. This in turn

[16] Legend shown in Table 1, page 5.

[17] Legend shown in Table 1, page 5.



(a) Active VCSTask extracting VCS metadata.

(b) Active Loop querying repository commits.

(c) Commit analysis of active MecoTask.

(d) Active MemeTask finishing the workflow.

Fig. 22 Software repository mining workflow execution. Green nodes represent active or deployed entities, gray nodes undeployed resources or scheduled tasks, orange tasks are skipped, and tasks with a green background and a gray icon are finished

starts the mongoQuery executable component querying all revisions of the project. The result is then stored in the decision input of the loop. After the VCSSensor has gathered the required decision-making information, it is stopped by the looped. As the analyzed projects contain at least one commit, the decision input is not empty resulting in the MemeTask to be skipped while the looped MecoTask remains in the scheduled state.

Finally, the loop iterates over each revision by providing the MecoTask with the revision to analyze in this iteration (Fig. 22c). This results in the deployment of the Meco application and both the Loop, as well as the MecoTask being in an active state. Even though the Loop is active its executable VCSSensor and the underlying monitoring component is removed as it is not required anymore. During this time step, the metrics of the revisions are analyzed and finally stored within the MongoDB database.

After analyzing each revision, the decision input of the Loop is empty resulting in a rescheduling of the MemeTask as well as the Loop task being finished. Therefore, the Meco application is deprovisioned and the MemeTask is ready to be executed (Fig. 22 (d)). Hereby, the Meme application is deployed followed by triggering the MemeTask which is now active. This in turn results in the execution of the meme-job deleting duplicates from the mined data stored in the database.

6.3.3 Parallel workflow execution

To speed up the software repository mining workflow, shown in Fig. 21, the MecoTask can be executed in parallel. There-

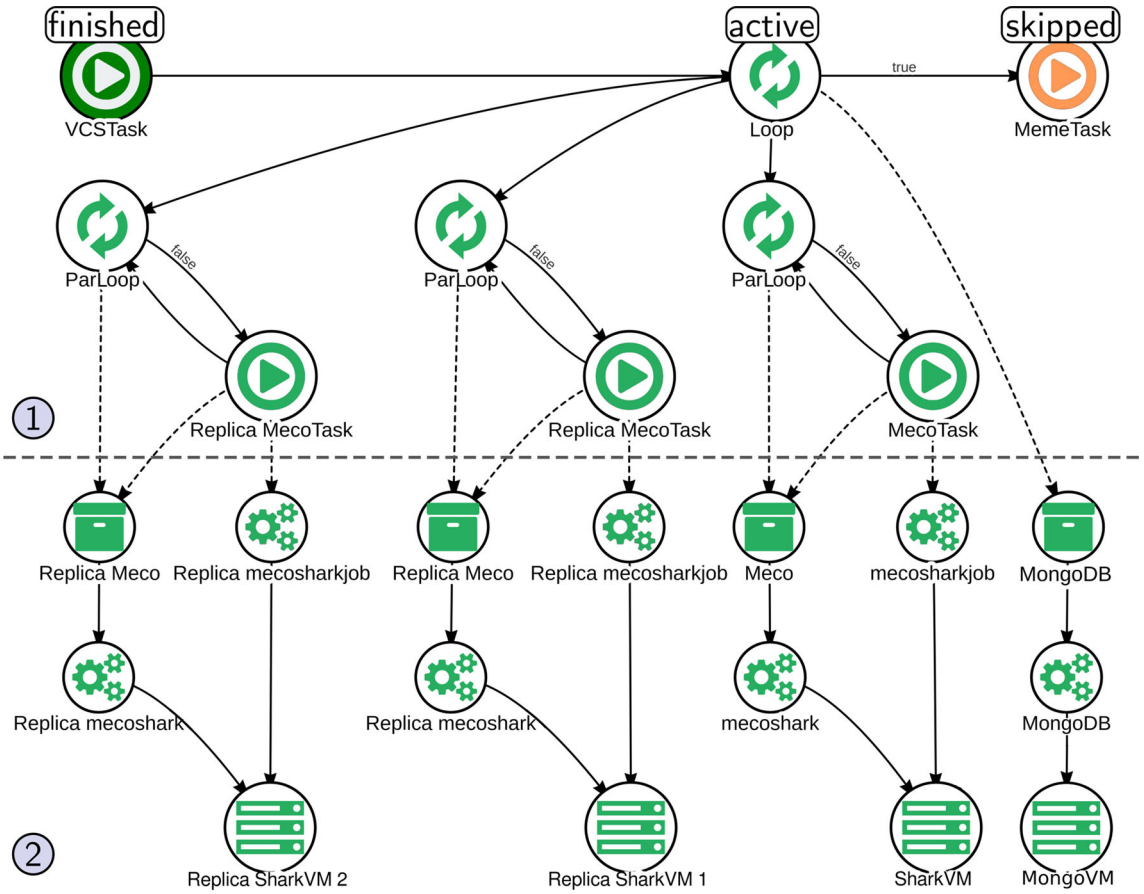


Fig. 23 Threefold parallelization of the repository mining loop

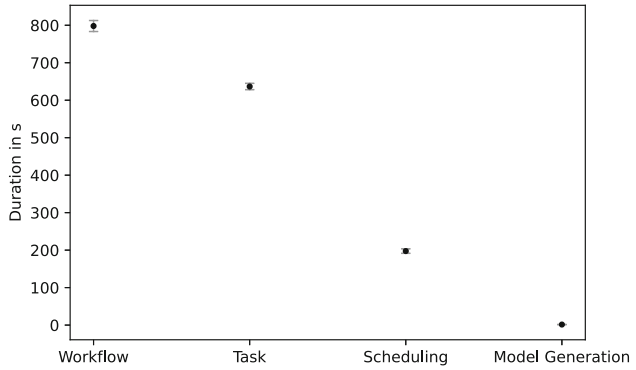
fore, we adjusted the workflow model by adding a parallel tag to the Loop task, as well as a shared tag to the MongoDB application. While the execution of the VCSTask and MemeTask remains the same when parallelizing the loop, the runtime state when looping over the MecoTask differs, as shown in Fig. 23.[18] In general, the activities taking place when the start action of a loop is triggered can be separated into two parts. The replication of elements on the workflow layer ① and the addition of required infrastructure ②.

The replication of workflow information ① includes the creation of the main (Loop) and three nested loops (ParLoop) which are connected over NestedDependency links. During the creation of nested loops, the main loop's decision input is already filled with the revision hashes to iterate over which are now evenly divided among the nested loops distributing the workload. In addition to the loop orchestration, the looped over MecoTask is replicated two times with the duplicates tagged as replica (Replica MecoTask).

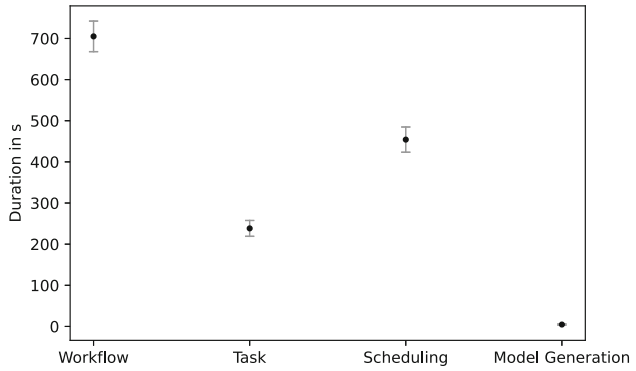
To add required infrastructure ②, the information of the original Loop and MecoTask is recreated by duplicating the sub graph with the target node of the platform dependency serving as root. Thus, each replica consists of its own Meco application including the mecoshark application component, as well as a mecosharkjob executable component which are hosted on a new VM (Replica SharkVM 1-2). As the MongoDB application is tagged as shared, this part of the subgraph is not duplicated which results in each mecosharkjob executable component to be connected to the same MongoDB. We omitted these links to the shared MongoDB for clarity.

During the execution of this phase in the workflow, the main loop observes its parallelized loops and gets notified when one of them has reached the finished state. Then, the main loop investigates whether all of its loops have been finished. After the completion of the Loop only the MongoVM and the original SharkVM remain, as it is reutilized for the MemeTask finalizing the workflow. Overall, the parallelization of the loops result in the addition of two additional VMs including the deployment of required applications to analyze the metrics of a part of the commits while storing the results within a shared database.

[18] Legend shown in Table 1, page 5.



(a) Non parallelized workflow execution.



(b) Three fold parallel workflow execution.

Fig. 24 Repository mining workflow 95% confidence interval plot for the gathered timing measurements in seconds for a project with 64 commits

6.3.4 Results and observations

Within this case study we showed that the decision-making process can be used as part of a loop to retrieve and iterate over a set of items by passing them to individual tasks. Furthermore, the case study demonstrated how an easy to parallelize problem can be solved using our approach.

At design time, we additionally created and generated a domain specific OCCI extension utilizing the MoDMaCAO framework. The extension introduces further attributes that can be configured within modeled components. The added attributes allow for more specific settings within the individual components, e.g., by providing an attribute that allows to set the repository to analyze. Due to the model-driven framework, this extension was easily modeled, generated and integrated within our process. The creation of the extension represents a one-time effort for scientists wanting to create more flexible and reusable elements within their workflow. Further extensions may enable, e.g., the deployment of a distributed and sharded MongoDB. Even though a further extension was used in this study, it did not affect the workflow's execution logic

At runtime and especially throughout the execution of the loop, we were able to monitor the current state of the infrastructure and workflow, as well as which commit is currently processed by the looped task. The parallelization of this loop showed that the tasks within the loops are processed at different pace due to the task's input. In the case study, e.g., the task execution depended on the size of the commit to analyze. This timing results in different kind of infrastructure compositions making it even more important to model platform dependencies correctly to minimize the deployment times. For example, if a platform dependency to the Meco application is modeled, the infrastructure of each replica group exist until all loops are finished, which results in excess resources. If no platform dependency is modeled, the infrastructure of a finished loop would be deprovisioned. However, this in turn may result in the deprovisioning of the SharkVM, which later in the workflow is required again. This issue can be coped with by improving the transformation process, e.g., by reutilizing replicated infrastructure. Even though the case study described an easy to parallelize problem, increasing the parallelization of the loop does not always speed up the overall workflow. When the loop is triggered multiple VMs spawn up simultaneously increasing the deployment time, especially on the small private cloud the workflow was tested on. The increased deployment time can result in a longer execution time of the overall workflow, specifically when only a small project is analyzed.

The timing measurements for the execution of the repository mining workflow on a project with 64 commits are visualized in Fig. 24 with the 95% confidence interval and mean for the non-parallelized execution and the threefold parallelized execution. The values for the non-parallelized execution (Fig. 24a) show a mean Workflow execution time of 798 s, Task execution time of 636 s, a Scheduling time of 197 s and runtime Model Generation time of 1.4 s. All values show a low variance. A majority of the Workflow execution time is taken by the Task execution. Compared to the parallel execution (Fig. 24b), this shifts to the Scheduling process taking up more time. This shift occurs due to the additional VMs and application deployments that need to be performed for the parallelization at runtime. While the overall Scheduling time rises to 454 s in the mean, the Task execution time sinks to 238 s in the mean, as the commits to analyze can be processed more quickly. Furthermore, in the parallelized workflow, a higher variance in the Task and Scheduling duration can be observed, which is likely related to the parallel management of workflow tasks and the time required to start multiple VMs in parallel. Also the sum of the Model Generation time rises to 4.6 s in the mean, which compared to the overall Workflow duration is still negligible.

Within a further experiment of the software repository mining workflow, we tested additional parallelization levels to analyze a repository of 272 commits using paralleliza-

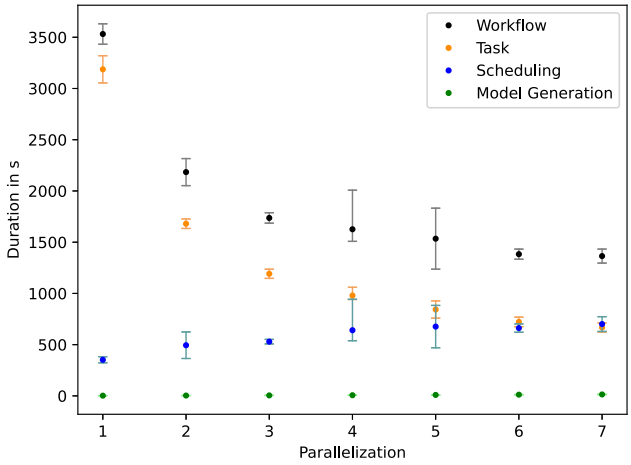


Fig. 25 95% Confidence intervals of repository mining workflow applied on a project with 272 commits using parallelization settings one to seven

tions from one to seven. In, Fig. 25, the 95% confidence interval of five executions for each parallelization setting is shown. The data show that an overall workflow speedup can be reached by parallelizing the task instances in the workflow, however, with the trade-off of an increased scheduling time. In case of this workflow execution, no significant speedup can be reached starting from parallelization three. Moreover, starting from parallelization five the scheduling and task execution times start to overlap. While this article highlights the possibility of integrating workflow parallelization into a runtime workflow model, the implementation of the proposed concept may be tweaked to improve the scheduling time. For example, by employing already deployed general-purpose hardware or containerized deployment strategies. The possibility of integrating containerization into the proposed concept is discussed in a Bachelor thesis [83] that was conducted in parallel with this study. In this experiment, the repository mining applications were already preinstalled within Docker containers instead of being deployed on VMs using configuration management. The first results are promising showing that the workflow deployment time of a SmartSHARK plugin could be reduced from 157 s, required by the configuration management, to 58 s by the containerized solution. Hereby, the 58 s mostly originate from downloading the one gigabyte large containers built to conduct the experiment. Overall, the experiment revealed that compute nodes can represent VMs, as well as container resources. Still, as containers hide the deployed components within them the reflection in the runtime model requires additional component instances in the model to reach the same amount of reflective capabilities.

7 Discussion

Within the case studies, we used the workflow runtime model as a knowledge base within a MAPE-K loop to schedule the required infrastructure for individual tasks at each point in time. Overall, the experiments show that runtime models are a great knowledge base for self-adaptive behavior and suitable to reflect and enact scientific workflows having dynamic and arbitrary infrastructure requirements. The case studies demonstrate that the execution of a workflow can be described via a sequence of declarative runtime states. These states help to monitor, analyze and plan the next required state which can then be enacted via the causal connection of the runtime model. Moreover, due to the utilization of a model representation of the system, the workflow and the underlying infrastructure can be easily simulated. Therefore, only the connector affecting the system has to be exchanged with, e.g., a dummy to simulate the required behavior when a VM is started or a task gets executed. This simulation allows the workflow model and scheduling process to be tested before an actual deployment. Additionally, this allows to not only statically validate modeled workflows but also investigate its execution behavior dynamically. This, in turn, supports the development process of scheduling mechanisms as they can be simulated and visualized using the runtime model. This simulation feature was used throughout the development of our SWFMS, as well as the example workflow models before an actual execution in the cloud environment.

In the following, we provide a more detailed discussion about the results observed in the case studies. In Sect. 7.1, we argue about the benefits and drawbacks of the workflow runtime model based on the perspectives of self-adaptive systems as well as human users. In Sect. 7.2, we discuss overheads introduced in our approach based on our observed measurements. Thereafter, in Sect. 7.3, lessons learned when working with workflow runtime models are covered and how execution times can be tweaked. Finally, the threats to validity are described in Sect. 7.4.

7.1 Workflow model perspectives

The pragmatic feature of a model [19] defines that the usefulness is partially dictated by the user of the model. In case of the workflow runtime model, we assume two kinds of users. The scientist modeling and observing the runtime state of the workflow and the cloud architect developing and providing the application deployments. Furthermore, we investigate the model perspective of self-adaptive control loops utilizing the runtime model as a knowledge base to automatically manage the workflow and its infrastructure.

From a scientists point of view, the runtime model allows to monitor, inspect and manage the state of their workflow and its infrastructure. Hereby, the states and attributes of each

element help to understand the overall running system. For example, the states allow to directly reflect failing deployments or tasks and actions allow to interact with the system, e.g., by starting or stopping individual tasks. Moreover, the causal connection supports human-in-the-loop activities as they can directly affect the decision-making process within the runtime model.

From a cloud architect point of view, the abstraction and causal connection of the runtime model to the system allows to interact and observe the provisioned infrastructure and deployed application. This direct connection allows the developer to design infrastructure deployments in an iterative manner while getting immediate feedback about designed component deployments. Especially the runtime information contained within the model provides the opportunity to design dynamic application components.

From a system point of view, self-adaptive control loops benefit the most from the detailed runtime information and the causal connection of the model to the system. In general, the self-adaptive control loops require a detailed representation of the system, while human users require an abstracted view of it. Thus, the more detail is modeled within the runtime model, the better suited it is for autonomous systems as they gain more knowledge about the running system and more ways to interact with it. This can be achieved, e.g., by increasing the used amount of sensors and their monitoring results. However, most of this information is not relevant to either the scientist or cloud architect requiring different views on the system. Still to maintain a causal connection to the system a certain degree of detail is required throughout the modeling process. While our approach combines the information contained within the design time and runtime model, it is worthwhile to discuss which of both models contains the more important information for the system. While the design time model describes the high-level goal to be achieved within the workflow, the runtime model may contain additional capabilities not intended by the design time model. In our approach, we value the information in the runtime model higher, as it allows other autonomic control loops to be attached to the process, e.g., to add scaling capabilities to databases. Within the repository mining process, for example, a component may be introduced implementing a control loop, which observes the size of the MongoDB and scales it accordingly.

To foster the utilization of different perspectives, different views on the model can be defined, e.g., by applying filters which hide resources of extensions representing individual layers. Thus, views can be defined highlighting resources of interest for the cloud architect, e.g., the infrastructure and platform layer, which can be hidden by the scientist. Also, the OCCI extension generation mechanisms of OCCIWare Studio allows for annotations on each introduced kind that can be reutilized on the visualization layer to incorporate, e.g., collapsing functionalities for applications and their components.

7.2 Design and runtime overhead

In this section, the overhead introduced by our approach is discussed covering design and runtime aspects.

At design time, our approach allows to create and share templates for frameworks and tooling required for the execution of specialized workflows. The creation of extensions that can be partially generated using the OCCIWare ecosystem allows the reutilization and runtime management of components, e.g., by affecting predefined attributes or performing actions. Even though the creation of these extensions requires knowledge about the cloud domain it represents a one-time effort that can be shared among the community. It should be noted that this knowledge is mainly related to the development of configuration management scripts, which are assigned to component instances. Hereby, the scripts can be developed in a standalone manner, as no specialized code is required to run them. Currently, few extensions and scripts for common cloud applications exist which we strive to extend in future work.

The runtime measurements of the individual timings of our approach revealed that the workflow execution time is dominated by the time required to execute the individual tasks, as well as the provisioning and deployment of the individual infrastructure. In the current implementation, the infrastructure is provisioned, as soon as it is required, which results in no task being executed at that time. Therefore, the overall workflow execution time is increased. To resolve this issue, predictions about the individual task execution times need to be incorporated into our approach allowing to spawn the infrastructure and application deployments ahead of time. Also, the utilization of further virtualization layers, such as containers, may greatly improve the deployment times. While our implementation can be improved to save cost and time, it should be noted that it is already more time efficient than the manual provisioning, deployment and configuration of applications.

Finally, maintaining a detailed representative state of the system within a runtime model may produce an overhead, especially when using heavily parallelized loops. For example, within the software repository mining workflow, each commit is analyzed one after another resulting in a correct reflection of the task currently running. Furthermore, for small projects in which each commit is analyzed within seconds a lot of requests and connections have to be made in comparison to a long running task. While the amount of requests could be reduced by batching up the commits to reduce the management overhead, a detailed reflection of the workflow is not granted anymore.

7.3 Lessons learned

To provide further insights about the utilization of the workflow runtime model this section shares some lessons learned during our experiences working with it.

While the workflow runtime model provides a causally connected representation of the system underneath, its level of detail correlates with the development effort put into the model. For example, to include certain monitoring capabilities like measuring the CPU utilization of a VM a sensor has to be added to the model, including its components and placements. This kind of information can be easily added using a model transformation on each required runtime model, e.g., by attaching the same kind of sensor to each VM. However, this comes with the drawback of increasing the size of the model cluttering its visual representation. To solve this issue, suitable perspectives need to be defined targeting different types of users as discussed in the previous section. Even though runtime metrics can be observed, they are limited to the reflection of one value at a time showing, e.g., an aggregate workload. Therefore, a history of observed metrics has to be recorded using an additional tool.

Independent of the perspective, the utilization of an abstract model to manage the infrastructure, as well as workflow proofed to be useful, as complex deployments could be easily modeled and connected to individual tasks. Moreover, the development of an architecture scheduling approach is supported by the runtime model, as declarative states can be generated through a model transformation and applied via a models at runtime engine. Especially as the causal connection of the runtime model to the cloud can be disabled to simulate and observe the developed scheduling behavior.

Understanding the scheduling mechanisms is crucial for modeling efficient workflows as the deployment and the configuration of resources is directly coupled with it. Especially platform dependencies need to be placed correctly to reduce the overall execution time of the workflow as deployment times can be drastically reduced. For example, in workflows containing short loop cycles the complete infrastructure may stay provisioned by attaching platform dependencies from the loop to all applications required during the loop task.

Providing access to the different layers of the workflow allows to tune its individual tasks. While the workflow layer allows, e.g., to affect the amount of parallel nested loops to speed up the execution, it does not directly affect the configuration of deployed applications. Still, due to the access to the platform elements, these frameworks can be modeled in such a manner that they parallelize and optimize the execution of a given task using the framework's logic. For example, in the case of the hadoop workflow, additional workers can be deployed or configurations can be adjusted to even further speed up the degree to which a MapReduce task is executed. Especially the addition of dedicated OCCI extensions would allow for a more flexible management of deployed applications, e.g., by providing up scaling actions that can be triggered to automatically add new worker nodes using a single request.

7.4 Threats to validity

Although we described only three scenarios to evaluate the feasibility of a workflow runtime model, we chose the case studies from different domains to examine its general applicability. The case studies are chosen to cover workflow models of different sizes. Furthermore, the case studies describe the main features of the workflow runtime model including its decision-making process, loops, and dataflows between distributed hosts. Moreover, the deployment of cloud applications using OCCI and the utilized adaptation engine was tested on further scenarios including common software stacks that may be used within workflows, e.g., *Linux, Apache, MySQL, PHP* (LAMP) for web services and *Elasticsearch, Logstash and Kibana* (ELK) for centralized logging [84].

To perform our case studies, we solely utilized VMs that got provisioned on a private OpenStack cloud. Even though the approach is only tested using a single cloud provider, the utilization of the OCCI cloud standard and its model-driven ecosystem allows to easily incorporate further provider-specific interfaces managed via the uniform interface of OCCI. Therefore, connectors to further cloud providers can be easily generated and implemented. Furthermore, we only utilized core functionalities provided by every IaaS cloud, i.e., the provisioning of VMs and networks. In the described scenarios, no containerization techniques are discussed which can reduce the overall deployment time during workflow execution. The concept of containers can be modeled with the runtime model and its implementation, as discussed by [75], and was combined with our approach in [83]. Moreover, a detailed evaluation of scheduling improvements based on clustered containerized resources, like in [61], or serverless functionality, like in [62], is missing and part of future work. Still, for the demonstration of the usefulness of runtime models for workflow execution the utilized virtualization technique is negligible.

In this article, we focus on the added benefits of coupling workflows with a runtime model as well as shifting user-defined infrastructures. Therefore, a systematic comparison to other approaches and SWFMS is not targeted and will be in the scope of a future article. To measure the time requirements of our approach, we performed a quantitative comparison of the individual procedures. We revealed that the model-driven portion of our workflow execution time is negligible and mostly depends on the provisioning and task execution, which can be optimized using improved application configurations or existing scheduling mechanisms for

cloud infrastructures. From a quantitative point of view, we delimited our approach to related work by highlighting the utilization of a runtime model build around an existing cloud standard. While we do not extensively compare the capabilities and functionalities offered by each SWFMS, we discuss the benefits of coupling a runtime model to workflows and cloud computing from different perspectives. Furthermore, by extending an existing cloud standard, we proved the concept of using OCCI as a datamodel and interface for the execution of workflows introducing a new layer to the standard.

8 Conclusion

In this paper, we described a workflow runtime model concept, which allows to reflect scientific workflows while maintaining a direct connection of individual tasks to their required infrastructure. Based on three case studies from different domains, we showed that our approach allows to dynamically fulfill arbitrary infrastructure requirements throughout the workflow execution. Moreover, we demonstrated that the runtime model serves as a prominent knowledge base for self-adaptive control loops to enact workflows by performing model transformations. To realize the concept, we utilized recent advances in infrastructure as code, deployment and modeling approaches and coupled them with workflow capabilities. The experiments revealed the usefulness of a workflow runtime model to reflect and enact scientific workflows. The causal connection of the runtime model as well as the simplicity of our approach fosters adjustments of the control flow or provisioned resources at runtime either automatically or by a human-in-the-loop.

In future work, we investigate suitable levels of abstraction to provide different views on the workflow runtime model to support the creation of workflow models at design time and personalize the observation of it at runtime. Additionally, we aim at including model-driven generation of general-purpose infrastructure removing the need to model compute nodes for simple tasks. Among others, this includes a detailed evaluation of the runtime model reflection of containerized resources and clusters as they reduce deployment times. Moreover, we aim to include the management of other computation resources such as high-performance computation clusters and internet-of-things devices within the runtime model. Finally, we plan to evaluate the applicability of our approach on extreme scale workflows by drastically increasing the amount of parallel loops while investigating its impact on the runtime model management. Hereby, we especially aim at conducting case studies that allow evaluating the extent to which the utilization of reflected infrastructure and communication channels can improve complex workflows.

Funding Open Access funding enabled and organized by Projekt DEAL.

Data Availability The implementation of our approach and videos demonstrating the example scenarios are publicly available at: https://gitlab.gwdg.de/rwm/smartwyrm.

References

1. Zhao, Y., Li, Y., Raicu, I., Shiyong, L., Tian, W., Liu, H.: Enabling scalable scientific workflow management in the cloud. Futur. Gener. Comput. Syst. **46**, 3–16 (2015)
2. Deelman, E., Gannon, D., Shields, M., Taylor, I.: Workflows and e-Science: an overview of workflow system features and capabilities. Futur. Gener. Comput. Syst. **25**(5), 528–540 (2009)
3. Liu, J., Pacitti, E., Valduriez, P., Mattoso, M.: A survey of data-intensive scientific workflow management. J. Grid Comput. **13**(4), 457–493 (2015)
4. Kovács, J., Kacsuk, P.: Occopus: a multi-cloud orchestrator to deploy and manage complex scientific infrastructures. J. Grid Comput. **16**(1), 19–37 (2018)
5. Qasha, R., Cała, J., Watson, P.: A framework for scientific workflow reproducibility in the cloud. In: Proceedings of the 12th IEEE International Conference on e-Science (e-Science) (2016)
6. Bencomo, Nelly, B., Gordon, Götz, S., Morin, B., Rumpe, B.: Report on the 7th International Workshop on Models@run.time. ACM SIGSOFT Softw. Eng. Notes **38**(1):27–30 (2013)
7. Szvetits, M., Zdun, U.: Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. Softw. Syst. Model. **15**(1), 31–69 (2016)
8. Bencomo, N., Götz, S., Song, H.: Models@run.time: a guided tour of the state of the art and research challenges. Softw. Syst. Model. **18**(5), 3049–3082 (2019)
9. Deelman, E., Peterka, T., Altintas, I., Carothers, C.D., Kleese, K., van Dam, K., Moreland, M.P., Ramakrishnan, L., Taufer, M., Vetter, J.: The future of scientific workflows. Int. J. High Perform. Comput. Appl. **32**(1), 159–175 (2018)
10. Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al.: Above the clouds: a berkeley view of cloud computing. Electrical Engineering and Computer Sciences, University of California at Berkeley (2009)
11. Open Grid Forum. Open Cloud Computing Interface - Core, 2016. Available online: https://www.ogf.org/documents/GFD.221.pdf, last retrieved: 05/05/2023
12. Erbel, J., Korte, F., Grabowski, J.: Scheduling architectures for scientific workflows in the cloud. In: Proceedings of the 10th International Conference on System Analysis and Modeling (SAM) (2018)

13. Erbel, J., Wittek, S., Grabowski, J., Rausch, A.: Dynamic management of multi-level-simulation workflows in the cloud. In: Proceedings of the 2nd International Workshop on Simulation Science (SimScience) (2019)
14. Object Management Group. Unified Modeling Language (2015). Available online: https://www.omg.org/spec/UML/2.5/PDF. Accessed 05 May 2023
15. Object Management Group. OMG: Business Process Model and Notation (2011). Available online: https://www.omg.org/spec/BPMN/2.0/PDF. Accessed 05 May 2023
16. Ludäscher, B., Weske, M., McPhillips, T., Bowers, S.: Scientific workflows: Business as usual?. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A., (eds.) Proceedings of the 7th International Conference on Business Process Management (BPM), pp. 31–47. Berlin, Heidelberg, Springer Berlin Heidelberg (2009)
17. Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E.A., Tao, J., Zhao, Y.: Scientific workflow management and the Kepler system. Concurr. Comput. Pract. Exp. **18**(10), 1039–1065 (2006)
18. Kühne, T.: Matters of (meta-) modeling. Softw. Syst. Model. **5**(4), 369–385 (2006)
19. Stachowiak, H.: Allgemeine Modelltheorie. Springer-Verlag, Berlin (1973)
20. Favre, J.M.: Towards a basic theory to model model driven engineering. In: Proceedings of the 3rd UML Workshop in Software Model Engineering (WiSME) (2004)
21. Blair, G., Bencomo, N., France, R.B.: Models@ run.time. Computer **42**(10), 22–27 (2009)
22. Kleppe, A.G., Warmer, J.B., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Professional, Boston (2003)
23. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electron. Notes Theor. Comput. Sci. **152**, 125–142 (2006)
24. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)
25. Mell, P., Grance, T.: The NIST Definition of Cloud Computing. National Institute of Standards and Technology, Gaithersburg (2011)
26. Organization for the Advancement of Structured Information Standards. TOSCA Simple Profile in YAML Version 1.3 (2020). Available online: https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.pdf. Accessed 05 May 2023
27. Wilde, M., Hategan, M., Wozniak, J.M., Clifford, B., Katz, D.S., Foster, I.: Swift: a language for distributed parallel scripting. Parallel Comput. **37**(9), 633–652 (2011)
28. Ramakrishnan, L., Poon, S., Hendrix, V., Gunter, D., Pastorello, G.Z., Agarwal, D.: Experiences with user-centered design for the Tigres workflow API. In: Proceedings of the 10th IEEE International Conference on e-Science (e-Science) (2014)
29. Altintas, I., Berkley C., Jaeger E., Jones, M., Ludascher, B., Mock, S.: Kepler: an extensible system for design and execution of scientific workflows. In: Proceedings of the 16th International Conference on Scientific and Statistical Database Management, 2004. pp. 423–424. IEEE (2004)
30. Barga, R., Jackson, J., Araujo, N., Guo, D., Gautam, N., Simmhan, Y.: The trident scientific workflow workbench. In: Proceedings of the 4th IEEE International Conference on e-Science (e-Science) (2008)
31. Bui, P., Yu, L., Thain, D.: Weaver: integrating distributed computing abstractions into scientific workflows using python. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (2010)
32. Churches, D., Gombas, G., Harrison, A., Maassen, J., Robinson, C., Shields, M., Taylor, I., Wang, I.: Programming scientific and distributed workflow with Triana services. Concurr. Comput. Pract. Exp. **18**(10), 1021–1037 (2006)
33. Deelman, E., Singh, G., Mei-Hui, S., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Karan Vahi, G., Berriman, B., Good, J., Laity, A., Jacob, J.C., Katz, D.S.: Pegasus: a framework for mapping complex scientific workflows onto distributed systems. Sci. Program. J. **13**(3), 219–237 (2005)
34. Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P.J., Mayani, R., Chen, W., Silva, R.F.D., Livny, M., et al.: Pegasus, a workflow management system for science automation. Futur. Gener. Comput. Syst. **46**, 17–35 (2015)
35. Goecks, J., Nekrutenko, A., Taylor, J., Team, G., et al.: Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. Genome Biol. **11**(8), 86 (2010)
36. Oinn, T., Greenwood, M., Matthew Addis, M., Alpdemir, N., Ferris, J., Glover, K., Goble, C., Goderis, A., Hull, D., Marvin, D., et al.: Taverna: lessons in creating a workflow environment for the life sciences. Concurr. Comput. Pract. Exp. **18**(10), 1067–1100 (2006)
37. Wolstencroft, K., Haines, R., Fellows, D., Williams, A., Withers, D., Owen, S., Soiland-Reyes, S., Dunlop, I., Nenadic, A., Fisher, P., Bhagat, J., Belhajjame, K., Bacall, F., Hardisty, A., Nieva, A., de la Hidalga, M.P., Vargas, B., Sufi, S., Goble, C.: The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. Nucleic Acids Res. **41**(W1), W557–W561 (2013)
38. Bavoil, L, Callahan, S.P., Crossno, P.J., Freire, J., Scheidegger, C.E., Silva, C.T., Vo, H.T.: Vistrails: enabling interactive multiple-view visualizations. In: Proceedings of the 16th IEEE Conference on Visualization (VIS) (2005)
39. Parker, S.G., Johnson, C.R.: SCIRun: a scientific programming environment for computational steering. In: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (SC) (1995)
40. Bergmann, R., Gil, Y.: Similarity assessment and efficient retrieval of semantic workflows. Inf. Syst. **40**, 115–127 (2014)
41. Craig Upson, T.A., Faulhaber, D.K., Laidlaw, D., Schlegel, D., Vroom, J., Gurwitz, R., Van Dam, A.: The application visualization system: a computational environment for scientific visualization. IEEE Comput. Graph. Appl. **9**(4), 30–42 (1989)
42. Tannenbaum, T., Wright, D., Miller, K., Livny, M.: Condor: a distributed job scheduler. In: Sterling, T. (ed.) Beowulf Cluster Computing with Windows, pp. 307–350. MIT Press, Cambridge (2001)
43. McGough, S., Young, L., Afzal, A., Newhouse, S., Darlington, J.: Workflow enactment in ICENI. In: Proceedings of the UK e-Science All Hands Meeting (2004)
44. Berman, F., Chien, A., Cooper, K., Dongarra, J., Foster, I., Gannon, D., Johnsson, L., Kennedy, K., Kesselman, C., Mellor-Crumme, J., et al.: The grads project: software support for high-level grid application development. Int. J. High Perform. Comput. Appl. **15**(4), 327–344 (2001)
45. Guan, Z., Hernandez, F., Bangalore, P., Gray, J., Skjellum, A., Velusamy, V., Liu, Y.: Grid-flow: a grid-enabled scientific workflow system with a petri-net-based interface. Concurr. Comput. Pract. Exp. **18**(10), 1115–1140 (2006)
46. Almond, J., Snelling, D.: Unicore: uniform access to supercomputing as an element of electronic commerce. Futur. Gener. Comput. Syst. **15**(5), 539–548 (1999)
47. Yu, J., Buyya, R.: A novel architecture for realizing grid workflow using tuple spaces. In: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID) (2004)
48. Fahringer, T., Jugravu, A., Pllana, S., Prodan, R., Seragiotto, C., Jr., Truong, H.L.: ASKALON: a tool set for cluster and grid computing. Concurr. Comput. Pract. Exp. **17**(2–4), 143–169 (2005)
49. Ogasawara, E., Dias, J., Silva, V., Chirigati, F., de Oliveira, D., Porto, F., Valduriez, P., Mattoso, M.: Chiron: a parallel engine

for algebraic scientific workflows. Concurr. Comput. Pract. Exp. **25**(16), 2327–2341 (2013)
50. Bouziane, HL, Pérez, C, Priol, T: A software component model with spatial and temporal compositions for grid infrastructures. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008–Parallel Processing, pp. 698–708. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
51. Aldinucci, M., Bouziane, H.L., Danelutto, M., Pérez, C.: STKM on SCA: a unified framework with components, workflows and algorithmic skeletons. In: Sips, H., Epema, D., Lin, H.X. (eds.) Euro-Par 2009 Parallel Processing, pp. 678–690. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
52. Vukojevic-Haupt, K., Haupt, F., Leymann, F.: On-demand provisioning of workflow middleware and services into the cloud: an overview. Computing **99**(2), 147–162 (2017)
53. Kacsuk, P., Kovács, J., Farkas, Z.: The flowbster cloud-oriented workflow system to process large scientific data sets. J. Grid Comput. **16**(1), 55–83 (2018)
54. Orzechowski, M., Balis, B., Pawlik, K., Pawlik, M., Malawski, M.: Transparent deployment of scientific workflows across clouds-kubernetes approach. In: Proceedings of the 11th IEEE/ACM International Conference on Utility and Cloud Computing (UCC) (2018)
55. Balis, B.: Hyperflow: a model of computation, programming approach and enactment engine for complex distributed workflows. Futur. Gener. Comput. Syst. **55**, 147–162 (2016)
56. Hoppe, D., Sandoval, Y., Sulistio, A., Malawski, M., Balis, B., Pawlik, M., Figiela, K., Krol, D., Orzechowski, M., Kitowski, J., et al.: Bridging the gap between HPC and cloud using Hyperflow and paasage. In: Proceedings of the 12th International Conference on Parallel Processing and Applied Mathematics (PPAM) (2017)
57. Achilleos, A.P., Kritikos, K., Rossini, A., Kapitsaki, G.M., Domaschka, J., Orzechowski, M., Seybold, D., Griesinger, F., Nikolov, N., Romero, D., et al.: The cloud application modelling and execution language. J. Cloud Comput. **8**(1), 20 (2019)
58. Weder, B., Breitenbücher, U., Képes, K., Leymann, F., Zimmermann, M.: Deployable self-contained workflow models. In: Proceedings of the 8th European Conference on Service-Oriented and Cloud Computing (ESOCC) (2020)
59. Breitenbücher, U., Endres, C., Képes, K., Kopp, O., Leymann, F., Wagner, S., Zimmermann, J.W.M.: The opentosca ecosystem -concepts & tools. Eur. Sp. Proj. Smart Syst. Big Data Future Internet Towards Serv. Grand Soc. Chall. **1**, 112–130 (2016)
60. Beni, E.H., Lagaisse, B., Joosen, W.: Infracomposer: policy-driven adaptive and reflective middleware for the cloudification of simulation & optimization workflows. J. Syst. Architect. **95**, 36–46 (2019)
61. Colonnelli, I., Cantalupo, B., Merelli, I., Aldinucci, M.: Streamflow: cross-breeding cloud with HPC. IEEE Trans. Emerg. Top. Comput. **9**(4), 1723–1737 (2021)
62. Roy, R.B., Patel, T., Gadepally, V., Tiwari, D.: Mashup: making serverless computing useful for HPC workflows via hybrid execution. In: Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '22. pp. 46-60. Association for Computing Machinery, New York (2022)
63. Open Grid Forum. Open Cloud Computing Interface - Platform (2016). Available online: https://www.ogf.org/documents/GFD.227.pdf. Accessed 05 May 2023
64. Erbel, J., Brand, T., Giese, H., Grabowski, J.: OCCI-compliant, fully causal-connected architecture runtime models supporting sensor management. In: Proceedings of the 14th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) (2019)
65. Korte, F., Challita, S., Zalila, F., Merle, P., Grabowski, J.: Model-driven configuration management of cloud applications with OCCI. In: Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER) (2018)
66. Open Grid Forum. Open Cloud Computing Interface - Infrastructure (2016). Available online: https://www.ogf.org/documents/GFD.224.pdf. Accessed 05 May 2023
67. Erbel, J., Korte, F., Grabowski, J.: Comparison and runtime adaptation of cloud application topologies based on OCCI. In: Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER) (2018)
68. Breitenbücher, U., Binz, T., Képes, K., Kopp, O., Leymann, F., Wettinger, J.: Combining declarative and imperative cloud application provisioning based on TOSCA. In: Proceedings of the 2nd IEEE International Conference on Cloud Engineering (IC2E) (2014)
69. Steinberg, D., Budinsky, F. (eds.): Merks, and Marcelo Paternostro. EMF, Eclipse Modeling Framework. Pearson Education (2008)
70. Paige, R.F., Kolovos, D.S., Rose, L.M., Drivalos, N., Polack, F.A.C.: The design of a conceptual framework and technical infrastructure for model management language engineering. In: Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on. pp. 162–171. IEEE (2009)
71. Merle, P., Barais, O., Parpaillon, J., Plouzeau, N., Tata, S.: A precise metamodel for open cloud computing interface. In: Proceedings of the 8th IEEE International Conference on Cloud Computing (CLOUD) (2015)
72. Open Grid Forum. Open Cloud Computing Interface - HTTP Protocol (2016). Available online: https://www.ogf.org/documents/GFD.223.pdf. Accessed 05 May 2023
73. Zalila, F., Challita, S., Merle, P.: A model-driven tool chain for OCCI. In: Proceedings of the 25th International Conference on Cooperative Information Systems (CoopIS) (2017)
74. Eclipse Foundation. Acceleo (2020). Available online: https://www.eclipse.org/acceleo/. Accessed 05 May 2023
75. Paraiso, F., Challita, S., Al-Dhuraibi, Y., Merle, P.: Model-driven management of docker containers. In: Proceedings of the 9th IEEE International Conference on Cloud Computing (CLOUD) (2016)
76. Apache Software Foundation. Hadoop (2020). Available online: https://hadoop.apache.org/. Accessed 05 May 2023
77. Apache Software Foundation. Spark (2020). Available online: https://spark.apache.org/. Accessed 05 May 2023
78. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
79. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST) (2010)
80. Wittek, S., Rausch, A.: Learning state mappings in multi-level-simulation. In: Proceedings of the 1st International Workshop on Simulation Science (SimScience) (2017)
81. Trautsch, F., Herbold, S., Makedonski, P., Grabowski, J.: Addressing problems with replicability and validity of repository mining studies through a smart data platform. Empir. Softw. Eng. **23**(2), 1036–1083 (2018)
82. Trautsch, A., Herbold, S., Grabowski, J.: A longitudinal study of static analysis warning evolution and the effects of PMD on software quality in apache open source projects. Empir. Softw. Eng. **25**(6), 5137–5192 (2020)
83. Thiesen, L.:Containerization in a causally connected runtime model for scientific workflows. Bachelor Thesis, 10:Institute of Computer Science. University of Goettingen, Germany (2020)
84. Challita, S., Korte, F., Erbel, J., Zalila, F., Grabowski, J., Merle, P.: Model-Based Cloud Resource Management with TOSCA and OCCI. Software and Systems Modeling (2021)

Johannes Erbel is a postdoctoral researcher at the Georg-August-Universität in Göttingen (Germany). He works in the research group Software Engineering for Distributed Systems at the Institute of Computer Science. Dr. Erbel's research interests focus on model-driven engineering approaches, cloud computing, and software evolution.

Jens Grabowski is professor at the Georg-August-Universität in Göttingen (Germany). He is heading the Software Engineering for Distributed Systems Group at the Institute. Prof. Grabowski is one of the developers of the standardized testing languages TTCN-3 and UML Testing Profile. The current research interests of Prof. Grabowski are directed towards model-based development and testing, managed software evolution, and empirical software engineering.